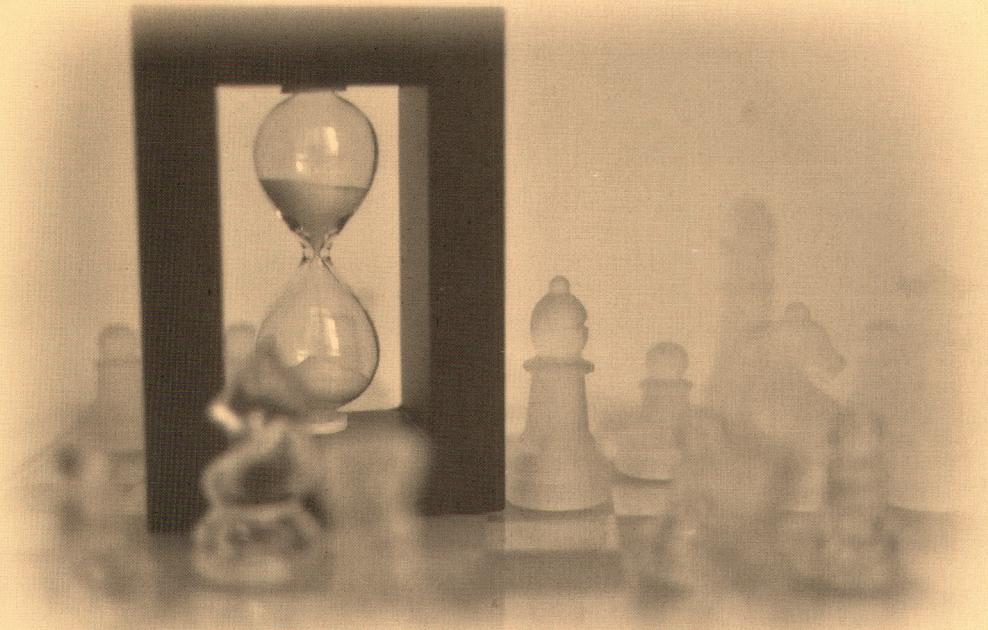


PROGRAMACIÓN Y ESTRUCTURAS DE DATOS AVANZADAS

**Lourdes Araujo Serna
Raquel Martínez Unanue
Miguel Rodríguez Artacho**



**Editorial Universitaria
Ramón Areces**



PROGRAMACIÓN Y ESTRUCTURAS DE DATOS AVANZADAS

LOURDES ARAUJO SERNA

Profesora Titular de Lenguajes y Sistemas Informáticos (UNED)

RAQUEL MARTÍNEZ UNANUE

Profesora Titular de Lenguajes y Sistemas Informáticos (UNED)

MIGUEL RODRÍGUEZ ARTACHO

Profesor Titular de Lenguajes y Sistemas Informáticos (UNED)

PROGRAMACIÓN Y ESTRUCTURAS DE DATOS AVANZADAS

Reservados todos los derechos.

Ni la totalidad ni parte de este libro puede reproducirse o transmitirse por ningún procedimiento electrónico o mecánico, incluyendo fotocopia, grabación magnética o cualquier almacenamiento de información y sistema de recuperación, sin permiso escrito de Editorial Centro de Estudios Ramón Areces, S.A. Diríjase a CEDRO (Centro Español de Derechos Reprográficos, www.cedro.org) si necesita fotocopiar o escanear algún fragmento de esta obra.

© EDITORIAL CENTRO DE ESTUDIOS RAMÓN ARECES, S.A.

Tomás Bretón, 21 - 28045 Madrid

Teléfono: 915.398.659

Fax: 914.681.952

Correo: cerala@cerasa.es

Web: www.cerala.es

ISBN-13: 978-84-9961-022-1

Depósito legal: M-27604-2011

Impreso por: Campillo Nevado, S.A.
Antonio González Porras, 35-37
28019 MADRID

Impreso en España / *Printed in Spain*

Índice

| | |
|---|------------|
| Índice de figuras | ix |
| Índice de tablas | xii |
| 1 Introducción | 1 |
| 1.1 El nacimiento de la programación | 1 |
| 1.2 Algoritmos y programación estructurada | 2 |
| 1.3 Esquemas algorítmicos | 3 |
| 1.4 Planificación del texto | 4 |
| 1.4.1 Prerrequisitos | 5 |
| 1.4.2 Dependencias entre capítulos | 5 |
| 1.4.3 Organización del contenido | 5 |
| 2 Estructuras de datos avanzadas | 9 |
| 2.1 Grafos | 9 |
| 2.1.1 Definiciones básicas | 10 |
| 2.1.2 Tipos de grafos | 11 |
| 2.1.3 Representación de grafos | 13 |
| 2.1.4 Recorrido de grafos | 17 |
| 2.1.5 Árboles de recubrimiento | 23 |
| 2.1.6 Puntos de articulación | 25 |
| 2.1.7 Ordenación topológica de un grafo dirigido acíclico | 27 |
| 2.1.8 Camino más corto desde la raíz a cualquier otro nodo | 29 |
| 2.1.9 Otros algoritmos sobre grafos | 30 |
| 2.2 Montículos | 30 |
| 2.2.1 Implementación y operaciones sobre elementos del montículo . | 33 |
| 2.2.2 Eficiencia en la creación de montículos a partir de un vector . | 40 |
| 2.2.3 Otros tipos de montículos | 43 |
| 2.3 Tablas de dispersión (<i>Hash</i>) | 45 |
| 2.3.1 Funciones Hash | 46 |
| 2.3.2 Resolución de colisiones | 50 |

| | |
|---|------------|
| 2.4 Ejercicios propuestos | 53 |
| 2.5 Notas bibliográficas | 55 |
| 3 Algoritmos voraces | 57 |
| 3.1 Planteamiento general | 57 |
| 3.1.1 Algoritmos voraces como procedimientos heurísticos | 63 |
| 3.2 Algoritmos voraces con grafos | 63 |
| 3.2.1 Árboles de recubrimiento mínimo: algoritmo de Prim | 64 |
| 3.2.2 Árboles de recubrimiento mínimo: algoritmo de Kruskal | 69 |
| 3.2.3 Camino de coste mínimo: algoritmo de Dijkstra | 71 |
| 3.3 Algoritmos voraces para planificación | 76 |
| 3.3.1 Minimización del tiempo en el sistema | 76 |
| 3.3.2 Planificación con plazos | 78 |
| 3.4 Almacenamiento óptimo en un soporte secuencial | 86 |
| 3.4.1 Generalización a n soportes secuenciales | 88 |
| 3.5 Problema de la mochila con objetos fraccionables | 89 |
| 3.6 Mantenimiento de la conectividad | 92 |
| 3.7 Problema de mensajería urgente | 93 |
| 3.8 Problema del robot desplazándose en un circuito | 96 |
| 3.9 Asistencia a incidencias | 101 |
| 3.10 Ejercicios propuestos | 102 |
| 3.11 Notas bibliográficas | 103 |
| 4 Divide y vencerás | 105 |
| 4.1 Planteamiento y esquema general | 105 |
| 4.2 Ordenación por fusión (<i>Mergesort</i>) | 109 |
| 4.3 El puzzle <i>tromino</i> | 110 |
| 4.4 Ordenación rápida (<i>Quicksort</i>) | 113 |
| 4.5 Cálculo del elemento mayoritario en un vector | 115 |
| 4.6 Liga de equipos | 117 |
| 4.7 <i>Skyline</i> de una ciudad | 120 |
| 4.8 Ejercicios propuestos | 124 |
| 4.9 Notas bibliográficas | 125 |
| 5 Programación dinámica | 127 |
| 5.1 Planteamiento general | 127 |
| 5.2 Los coeficientes binomiales | 130 |
| 5.3 Devolución de cambio | 132 |
| 5.4 El viaje por el río | 137 |
| 5.5 La mochila | 141 |
| 5.6 Multiplicación asociativa de matrices | 143 |
| 5.7 Camino de coste mínimo entre nodos de un grafo dirigido | 148 |

| | | |
|---------------------|--|------------|
| 5.8 | Distancia de edición | 152 |
| 5.9 | Ejercicios propuestos | 155 |
| 5.10 | Notas bibliográficas | 157 |
| 6 | Vuelta atrás | 159 |
| 6.1 | Planteamiento general | 159 |
| 6.2 | Coloreado de grafos | 167 |
| 6.3 | Ciclos Hamiltonianos | 169 |
| 6.4 | Subconjuntos de suma dada | 172 |
| 6.5 | Reparto equitativo de activos | 174 |
| 6.6 | El robot en busca del tornillo | 177 |
| 6.7 | Asignación de cursos en una escuela | 180 |
| 6.8 | Ejercicios propuestos | 182 |
| 6.9 | Notas bibliográficas | 184 |
| 7 | Ramificación y poda | 185 |
| 7.1 | Planteamiento general | 185 |
| 7.2 | Asignación de tareas: Pastelería | 195 |
| 7.3 | El viajante de comercio | 199 |
| 7.4 | Selección de tareas: cursos de formación | 203 |
| 7.5 | Distancia de edición | 207 |
| 7.6 | Ejercicios propuestos | 212 |
| 7.7 | Notas bibliográficas | 214 |
| Bibliografía | | 216 |
| A | Notación | 221 |

Índice de figuras

| | | |
|------|--|-----|
| 2.1 | <i>Grafo no dirigido</i> | 10 |
| 2.2 | <i>Grafo dirigido</i> | 10 |
| 2.3 | <i>Ejemplos de grafos</i> | 12 |
| 2.4 | <i>Grafo no dirigido con dos componentes conexas</i> | 12 |
| 2.5 | <i>Grafo dirigido con dos componentes fuertemente conexas</i> | 13 |
| 2.6 | <i>Ejemplo de recorrido en profundidad de un grafo</i> | 19 |
| 2.7 | <i>Ejemplo de recorrido en anchura de un grafo</i> | 23 |
| 2.8 | <i>Ejemplo de árbol de recubrimiento a partir de un recorrido en profundidad</i> | 24 |
| 2.9 | <i>Ejemplo de bosque de recubrimiento a partir de un recorrido en profundidad</i> | 25 |
| 2.10 | <i>Ejemplo de grafo con puntos de articulación</i> | 25 |
| 2.11 | <i>Ejemplo de grafo con puntos de articulación y su árbol de recubrimiento asociado</i> | 27 |
| 2.12 | <i>Ejemplo de grafo con los valores de numOrden[] y bajo[] calculados</i> | 27 |
| 2.13 | <i>Un grafo dirigido acíclico</i> | 28 |
| 2.14 | <i>Un grafo dirigido acíclico</i> | 29 |
| 2.15 | <i>Un grafo dirigido infinito</i> | 30 |
| 2.16 | <i>Estructura básica de un montículo</i> | 31 |
| 2.17 | <i>Un montículo de máximos M</i> | 32 |
| 2.18 | <i>M tras añadir 7</i> | 32 |
| 2.19 | <i>M tras eliminar la cima</i> | 32 |
| 2.20 | <i>Operación flotar</i> | 36 |
| 2.21 | <i>Operación hundir</i> | 37 |
| 2.22 | <i>Operación de la recuperación de la propiedad de montículo tras el borrado de la cima</i> | 39 |
| 2.23 | <i>Un montículo binomial compuesto por los subárboles M₀, M₁ y M₂</i> | 43 |
| 2.24 | <i>Un montículo de fibonacci</i> | 44 |
| 2.25 | <i>Una función Hash para una agenda</i> | 45 |
| 3.1 | <i>Grafo conexo no dirigido con pesos</i> | 67 |
| 3.2 | <i>Árbol de recubrimiento mínimo</i> | 68 |
| 3.3 | <i>Grafo conexo dirigido con pesos</i> | 73 |
| 4.1 | <i>Rotaciones de un tromino</i> | 110 |

ÍNDICE DE FIGURAS

| | | |
|-----|--|-----|
| 4.2 | <i>Retículo con casilla en negro</i> | 111 |
| 4.3 | <i>Proceso de resolución en primera recursión</i> | 111 |
| 4.4 | <i>Proceso de resolución en segunda recursión</i> | 112 |
| 4.5 | <i>Una ciudad, con edificios</i> | 120 |
| 4.6 | <i>El skyline de la ciudad anterior</i> | 121 |
| 4.7 | <i>Una ciudad, con 4 edificios</i> | 122 |
| 4.8 | <i>Línea de horizonte resultante</i> | 124 |
| 5.1 | <i>Ejemplo de una serie de embarcaderos A, B, \dots, N por el río. Los arcos indican los costes de los distintos trayectos.</i> | 137 |

Índice de tablas

| | | |
|-----|--|-----|
| 3.1 | <i>Matriz de adyacencia del circuito del robot</i> | 99 |
| 3.2 | <i>Traza de la particularización del algoritmo de Dijkstra al circuito del robot</i> | 100 |

Capítulo 1

Introducción

Este capítulo introduce el concepto de la algoritmia desde un enfoque histórico y propone un contexto de estudio y los requisitos aconsejables para un buen aprovechamiento del texto.

1.1 El nacimiento de la programación

El desarrollo de programas como disciplina nace tras la aparición de los primeros lenguajes de programación en la década de los 50. En esta primera fase, la programación era dependiente de los procesadores y los conceptos de control de flujo y las estrategias algorítmicas eran bastante difusos. En esta fase temprana, la producción de programas estaba basada en la prueba y error, y las estrategias para la resolución algorítmica de ciertos problemas no tenían todavía un lenguaje de especificación de código con suficiente nivel de abstracción. Sin embargo, entre 1950 y 1960 aparecen varios lenguajes de programación como ALGOL y FORTRAN que permiten identificar elementos de diseño algorítmico, muchos de los cuales han pervivido hasta nuestros días.

Más tarde, a finales de la década de los sesenta, la forma tradicional de producción de los programas en base a un primer diseño seguido de una batería de pruebas con juegos de datos de diversos tipos, fue cuestionada fuertemente y se constató la falta de una metodología de desarrollo y verificación de los programas que hacía que éstos no fueran fiables, lo que constituyó una barrera importante para el arte de la programación.

Como consecuencia de estas reflexiones surgen en aquel momento dos líneas de investigación paralelas:

- Una línea de desarrollo de los métodos de programación que produjeran procesos manejables mentalmente y por tanto sencillos de entender y validar.
- Una línea de desarrollo de métodos de verificación de programas cuyo objeto era la formalización de procedimientos que permitieran decidir que un programa sa-

tisface las especificaciones por estudio de su estructura y no por las técnicas de prueba.

La primera de ellas introdujo la noción de programación estructurada, el concepto de módulo y, con el tiempo, la noción de objeto; la segunda desarrolló las bases axiomáticas para la verificación de programas y también en menor medida para la producción de programas en función de una especificación previa.

1.2 Algoritmos y programación estructurada

La programación estructurada constituyó una traslación a la programación de las técnicas generales de resolución de problemas (mecanizadas a partir de la idea de N. Simon y Newell con el General Problem Solver [NS63]) pero también, a partir de estas ideas, comienza a evolucionar la noción del diseño de programas como un proceso de ingeniería.

Las aportaciones de Edgser W. Dijkstra, C.A.R. Hoare, O.J. Dahl y N. Wirth, entre otros, introdujeron a finales de los años 60 y principios de los 70 la noción de programación *estructurada* [Dij68, Dij72, Wir71, DDH72], planteando abstracciones estructurales en el código, que hasta entonces se veía como una mera colección lineal de asertos, lo que supuso la primera evolución hacia el desarrollo disciplinar de programas. Fue con estas abstracciones como la la programación modular y las técnicas de descomposición de módulos desarrollada por D.L. Parnas [GP70, Par72] como se produjo la primera revolución conceptual de la programación, precursora de la segunda que fue la introducción de la programación orientada a objetos, seguida a su vez por la programación basada en patrones de diseño.

A comienzos de los setenta se gesta también otra revolución conceptual, esta vez acerca de la representación y especificación de tipos de datos. Alrededor del concepto de *tipo abstracto de datos* [Gut75] evolucionan las ideas de modularidad, genericidad, esquemas genéricos de resolución de problemas, formalización del concepto de implementación y especificación formal de tipos de datos; ideas, todas ellas, armónicas con el diseño descendente y la programación estructurada. Se trata de distinguir entre la especificación de un tipo de datos, o su comportamiento observable, y sus implementaciones. Los mecanismos de encapsulamiento para aislar el acceso y manipulación de tipos de datos de su implementación se incorporaron rápidamente a lenguajes de programación como Pascal y Modula-2 [Wir82] entre otros.

También a mediados de los 70 la algorítmica adquiere forma de disciplina dominante en el campo de la informática con el desarrollo de muchos lenguajes de alto nivel y de métodos formales de desarrollo, que trajeron también la necesidad de analizar formalmente la complejidad de los algoritmos con la introducción del concepto de tratabilidad y de la distinción entre el crecimiento exponencial y el crecimiento polinomial por los trabajos de A. Cobham y J Edmonds [Cob65]. Finalmente es Knuth quien introduce la

notación O , Θ y Ω [Knu76], la cual permite facilitar enormemente el estudio formal de la eficiencia de los algoritmos, y que ha llegado hasta nuestros días.

Otro de los campos que han influido en la algorítmica aunque en menor medida es el de la verificación de programas, que viene a recoger la aserción de Edger W. Dijkstra de que los casos de prueba sirven para detectar la presencia de errores, pero no para asegurar su ausencia. En esta línea de razonamiento y buscando métodos de validación sistemática se posicionaron a mediados de los años 60 John McCarthy, P. Naur y sobre todo R. Floyd [McC63, Flo67]. Aunque el origen de la noción de verificación se puede remontar a Alan Turing [Tur49], fue Robert Floyd el primero en proponer el desarrollo de un método sistemático de verificación de algoritmos con la publicación en 1967 de su artículo “*Assigning Meanings to Programs*”, en el que se expone la idea de etiquetar en forma de asertos lógicos las sentencias de los programas de forma que se reflejaran los efectos de las mismas a partir de una definición formal de la semántica del lenguaje de programación. Fue poco después C. Hoare quien en 1969 desarrolló a partir de los trabajos de Floyd el cálculo semántico de pre y postcondiciones en algoritmos y el concepto de *invariante* en un bucle [Hoa69]. De estos trabajos desarrollados para la verificación de algoritmos deterministas surgió lo que se conoce como *Lógica de Hoare* [Krz81], a cuyo posterior desarrollo y ampliación contribuyeron entre otros N. Wirth y D. Gries [Gri82]. Estas reglas axiomáticas de prueba propuestas tuvieron pronto una enorme acogida que se amplió hacia la programación concurrente y a la programación distribuida [OG76, Owi76, Krz86].

1.3 Esquemas algorítmicos

La idea de que una familia de problemas pueden compartir una solución algorítmica eficiente adquiere gran interés en esta época, donde abundan las publicaciones con soluciones a determinados problemas abordables computacionalmente como los algoritmos de ordenación [Hoa62, Wil64], problemas de caminos mínimos en grafos y un largo etcétera. Como consecuencia de lo anterior, se introduce poco a poco la idea de que es posible agrupar determinados problemas algorítmicos bajo un esquema de resolución común.

Es en los años 60 donde se menciona por primera vez el concepto de esquema algorítmico que aparece desarrollado en la obra de Knuth “*The Art of Programming*”, auténtica biblia de la algorítmica comenzada en 1970 con la ayuda de Floyd (también en el CS Department, en Stanford) quien ayudó en las primeras revisiones, y continuada y ampliada hasta la actualidad con un total de 7 volúmenes, algunos de ellos todavía en renovación. Posteriormente se acuñan determinados nombres para las familias de algoritmos desarrolladas hasta el momento; así, la definición de algoritmo *voraz* se debe a Jack Edmonds [Edm71]. El estudio del problema de la ordenación, con los desarrollos de los algoritmos de ordenación rápida y la creación de las estructuras de datos de *montículos* para la solución de Williams [Hoa62, Wil64] se apoyan en la descomposición algorítmica.

mica que dio lugar al esquema de tipo Divide y Vencerás. Por otro lado, la descomposición en subejemplares solapados derivó en las técnicas de programación dinámica desarrolladas por Bellman, Floyd y Warshall, entre otros. El nombre del esquema conocido como *Divide y Vencerás* se atribuye a J. Mauchly, impulsor del proyecto de construcción de la ENIAC y uno de los fundadores de la ACM.

La ampliación de los esquemas iniciales, restringida a la resolución de los problemas de caminos mínimos y árboles de expansión mínimos en grafos explícitos y de ordenación, se complementa con la acuñación del concepto de Inteligencia Artificial por John McCarthy, que añadió a mediados de los 60 la investigación en algoritmos de búsqueda en grafos, búsquedas en anchura y en profundidad, y técnicas como el método de *minimax* y el algoritmo *A**, entre otros. Los algoritmos de ramificación y poda también son fruto de estas investigaciones y motivados principalmente con los problemas que plantea la Investigación Operativa. Entre la década de los 60 y comienzos de los 70 años, por ejemplo el *backtracking* o vuelta atrás se describe en 1965 por [GB65] con una solución algorítmica al problema de las ocho reinas.

Los *esquemas algorítmicos*, van cristalizando poco a poco y se asientan sus nombres y propiedades como elementos básicos de la computación. La importancia de los esquemas no se limita a la creación de soluciones genéricas a familias de problemas, y la evolución de éstos ha permitido, no solo la solución algorítmica de determinados problemas conocidos, sino la aparición de soluciones computables (tratables) de otros. Un ejemplo conocido de esto último es la FFT (transformada rápida de Fourier) que ha posibilitado el desarrollo del procesamiento digital de señales en tiempo real [BM67].

A partir de la década de los 70 la mayoría de los desarrollos realizados en esta línea se introducen de manera indiscutible como parte esencial de los currícula de la disciplina informática, que adquiere personalidad propia con la creación de las primeras facultades. De esta manera, el estudio de los esquemas algorítmicos y el conocimiento de sus múltiples aplicaciones ha llegado hasta nuestros días como un componente básico de la formación del ingeniero en informática.

1.4 Planificación del texto

Este libro está pensado como texto de apoyo a la enseñanza de la algorítmica en escuelas técnicas superiores de ingeniería informática.

Los contenidos combinan el estudio de algunas estructuras de datos y de los principales esquemas algorítmicos, junto con ejercicios y soluciones de los mismos. Cada capítulo, a su vez, se estructura con un desarrollo del tema que aborda seguido de ejercicios resueltos, propuestos y de notas bibliográficas. Hemos querido también realizar una presentación del tema al comienzo de cada capítulo para que el lector pueda conocer las dificultades que implica su estudio y los conocimientos previos requeridos.

1.4.1 Prerrequisitos

En primer lugar se supone una base matemática que implica entre otras cosas capacidad para la resolución de ecuaciones básicas y la realización de demostraciones por inducción. La base matemática se entiende fundamentada en los conocimientos habituales adquiridos en un primer curso de ingeniería e incluiría el cálculo de expresiones combinatorias básicas, nociones de probabilidad, cálculo de límites, polinomios, resolución de sistemas de ecuaciones lineales, y manejo de la notación matemática asociada.

En el aspecto de análisis algorítmico, se hará uso en mayor medida del cálculo y demostración de costes algorítmicos, por lo que puede necesitarse realizar demostraciones por reducción al absurdo o por inducción para la comprobación de optimalidad de determinados algoritmos y el manejo de sumatorios, combinatoria y cálculo de límites.

El texto da por conocidas las técnicas de análisis de coste de algoritmos y las notaciones O y Ω , aunque se usarán en muchas ocasiones las expresiones resueltas de las ecuaciones de recurrencia aplicadas a los algoritmos recursivos. Se suponen también conocidas las estructuras de datos básicas como listas, colas, árboles y sus operaciones básicas.

1.4.2 Dependencias entre capítulos

El libro se estructura en dos partes: estructuras de datos y esquemas de algorítmica. La primera de ellas expone estructuras de datos con un objetivo práctico. Se estudiará la utilidad de los mismos en el ámbito de la implementación en algoritmos de diverso tipo. Por esta razón este capítulo hace hincapié en algunas implementaciones eficientes de los mismos y se realizan referencias a los ámbitos en los que se usan.

En segundo lugar, se abarca el estudio de los esquemas algorítmicos con ejemplos de aplicaciones en diversos ámbitos, como la planificación, la ordenación y la búsqueda, entre otros. Todos los capítulos de algorítmica usan en mayor o menor medida conceptos de estructuras de datos, por lo que se recomienda que éste capítulo se abarque en su totalidad antes de proceder al estudio de alguno de los esquemas.

En cuanto al orden de estudio de los esquemas algorítmicos, se recomienda abordar por un lado los algoritmos voraces antes que la programación dinámica, por otro el estudio de los algoritmos de tipo divide y vencerás, y por último el de vuelta atrás antes que el de ramificación y poda.

1.4.3 Organización del contenido

Desde el punto de vista del profesor, el contenido y la planificación que proponemos es la siguiente:

- El Capítulo 2 abarca el estudio de las estructuras de datos avanzadas necesarias para el desarrollo de los algoritmos y esquemas expuestos.

- El Capítulo 3 incluye el estudio de los algoritmos voraces. El estudio de este esquema se basa en el conocimiento de aplicaciones conocidas del mismo, sobre todo involucrando problemas con grafos (como el del árbol de recubrimiento mínimo o el de los caminos mínimos) y de planificación de tareas. Se recomienda empezar con la introducción teórica al esquema y algún ejercicio y posteriormente se abordarían las diferentes familias de problemas que se resuelven mediante este esquema: grafos, planificación y optimización en varias variantes. Es recomendable que el estudio se realice cuando se tienen recientes los conocimientos sobre grafos y montículos.
- El Capítulo 4 abarca el estudio de los algoritmos de tipo Divide y Vencerás. En este tipo de esquema, la variedad de subfamilias es menor. Debe hacerse más hincapié en la inducción realizada en las llamadas recursivas, y por tanto que el aspecto teórico del esquema se trabaje bien antes de abordar la instanciación. Se comenzaría con el estudio teórico y después se abordarían dos o tres problemas en profundidad que ilustren la instanciación del esquema.
- El Capítulo 5 estudia la resolución de problemas mediante programación dinámica. Es conveniente conocer previamente los esquemas de divide y vencerás y de algoritmos voraces, por utilizar la programación dinámica conceptos algorítmicos de ambos.
- El Capítulo 6 contempla el estudio de los algoritmos de *backtracking* o de retroceso. Habría que repasar el concepto de grafo y abordar el esquema exponiendo claramente el concepto de exploración en un grafo implícito frente al concepto de grafo explícito de la estructura de datos. De esta manera, es importante que se adquiera conciencia de que por un lado la exhaustividad, y por otro la generación implícita del recorrido son los aspectos fundamentales, además claro está, de la capacidad de retroceso en el espacio de búsqueda.
- Por último, en el Capítulo 7 y como variante de lo anterior, el esquema de ramificación y poda propone una búsqueda con objetivo de optimización. Se debe repasar la noción de montículo para el almacenamiento de los nodos pendientes de desarrollo y detallar porqué esta estructura de datos proporciona un mecanismo de ramificación basada en expandir en cada momento el nodo más prometedor. En los aspectos teóricos este esquema es similar al anterior, por lo que las explicaciones deben abordarlo como una variante e identificar los elementos nuevos, como son el montículo, ya mencionado, y la actualización de las cotas superiores e inferiores del objetivo que se precisa optimizar.

El texto se complementa con bibliografía en donde se profundiza en algunos aspectos de la algoritmia relacionados con la eficiencia o con variantes de determinados problemas, así como con la realización de pruebas de evaluación personal al concluir el estudio

de cada esquema. Por último es recomendable complementar el estudio teórico con la realización de prácticas en las que se aborde la resolución y codificación completa de los esquemas mencionados.

Capítulo 2

Estructuras de datos avanzadas

Este capítulo está dedicado al estudio de algunas estructuras de datos avanzadas, en concreto: los grafos, los montículos y las tablas *hash*. Tanto los grafos como los montículos se utilizarán como estructuras fundamentales en algunos esquemas algorítmicos que se verán posteriormente.

El principal objetivo de este capítulo con respecto a los grafos y los montículos, es que el lector sea capaz de utilizarlos de forma adecuada y eficiente en los esquemas algorítmicos en los que sean necesarios. Para ello, debe entender sus fundamentos y conocer las operaciones básicas. Con respecto a las tablas *hash*, aunque su estudio y aplicación se limita a este capítulo, se pretende que el lector comprenda su utilidad y conozca las principales funciones *hash* y de resolución de colisiones asociadas a esta estructura de datos.

Se recomienda al lector una lectura secuencial del capítulo. Cuando vaya a abordar los problemas resueltos, se propone al lector que realice la traza del algoritmo propuesto sobre la estructura de datos, con el fin de comprender su modo de operar y funcionalidad. Finalmente, el lector puede intentar resolver los problemas propuestos, sabiendo que algunas de sus soluciones se encuentran en diferentes fuentes bibliográficas a las que se hace referencia al final del capítulo.

2.1 Grafos

Un grafo es una colección de nodos o vértices unidos por líneas o aristas. Se puede definir por lo tanto como la pareja $G = \langle N, A \rangle$, donde N es un conjunto finito de vértices o nodos y A es un conjunto finito de líneas o aristas, tal que cada línea o arista es un par de la forma $A \in N \times N$. En la figura 2.1 se puede observar un ejemplo de un grafo.

En general, los grafos permiten modelar problemas en los que existe una relación relevante entre los objetos que intervienen. Los nodos o vértices representarían los objetos y las aristas las relaciones entre ellos.

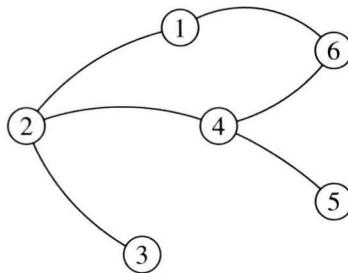


Figura 2.1: Grafo no dirigido

2.1.1 Definiciones básicas

Si las líneas o aristas en A están orientadas, es decir indican un sentido, se denominan *flechas* o *arcos* y al grafo se le denomina *grafo dirigido*. Si las líneas no están orientadas al grafo se le denomina *grafo no dirigido*. El grafo de la figura 2.1 es un grafo no dirigido. Una arista que une dos nodos n_1 y n_2 en un grafo dirigido se representará mediante el par ordenado $\langle n_1, n_2 \rangle$, indicando que el sentido de la flecha va del nodo n_1 al nodo n_2 . Por lo tanto $\langle n_1, n_2 \rangle$ y $\langle n_2, n_1 \rangle$ representan dos aristas distintas. Mientras que una arista que une los mismos nodos en un grafo no dirigido se representará por el conjunto (n_1, n_2) , y así (n_1, n_2) y (n_2, n_1) representan la misma arista.

El grafo de la figura 2.1 se describiría: $G = \langle N, A \rangle$, siendo $N = \{1, 2, 3, 4, 5, 6\}$ y $A = \{(1, 2), (1, 6), (6, 4), (4, 5), (2, 3), (2, 4)\}$.

En la figura 2.2 se puede ver un ejemplo de grafo dirigido. Este grafo se describiría: $G = \langle N, A \rangle$, siendo $N = \{1, 2, 3, 4, 5, 6\}$ y $A = \{\langle 2, 1 \rangle, \langle 1, 6 \rangle, \langle 6, 4 \rangle, \langle 4, 5 \rangle, \langle 3, 2 \rangle, \langle 2, 4 \rangle, \langle 4, 4 \rangle\}$.

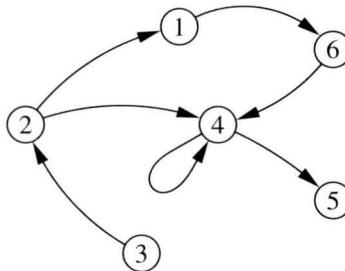


Figura 2.2: Grafo dirigido

Dado el arco $\langle n_i, n_j \rangle$, n_i es el *extremo inicial* del arco y n_j es el *extremo final*. A una arista de la forma (n_i, n_i) se la denomina *bucle* o *lazo*. El arco $\langle 4, 4 \rangle$ del grafo de la figura 2.2 es un bucle.

Dos nodos n_i y n_j son *adyacentes* si siendo distintos existe una arista o arco que los une. En los dos grafos ejemplo, tanto el de la figura 2.1 como el de la figura 2.2, los nodos adyacentes al 6 son el 1 y el 4.

El *grado* de un vértice en un grafo no dirigido es el número de aristas que salen o entran de él. Si el grafo es dirigido se puede hablar de *grado de entrada* de un vértice o *entrante* de un vértice, que es el número de arcos que llegan a él, y de *grado de salida* o *saliente* de un vértice, que es el número de arcos que salen de él. En el grafo no dirigido de la figura 2.1 el grado del vértice 2 es 3. En el grafo dirigido de la figura 2.2, el grado de entrada del vértice 2 es 1 y el grado de salida es 2.

Un *camino* en un grafo dirigido es una secuencia finita de arcos entre dos vértices, tal que el extremo final de cada arco coincide con el extremo inicial del arco siguiente. Si el grafo es no dirigido, un camino es una secuencia de aristas consecutivas. La *longitud de un camino* es el número de aristas que contiene. Un *camino simple* es un camino que no utiliza más de una vez la misma arista; en caso contrario se denomina *camino compuesto*. En el grafo de la figura 2.2, la secuencia de arcos $\langle 3, 2 \rangle, \langle 2, 1 \rangle, \langle 1, 6 \rangle$, es un camino simple de longitud 3. En el grafo de la figura 2.1 la secuencia de aristas $(3, 2), (2, 4), (4, 6)$ es un camino simple; pero dicha secuencia no es un camino en el grafo dirigido de la figura 2.2 ya que no existe el arco $\langle 4, 6 \rangle$. En el grafo de la figura 2.1 el camino $(3, 2), (2, 4), (4, 2)$ no es un camino simple sino compuesto, ya que en un grafo no orientado las aristas $(2, 4)$ y $(4, 2)$ son la misma arista y, por lo tanto, se está utilizando más de una vez la misma arista.

Un *ciclo o circuito* es un camino simple que empieza y termina en el mismo vértice. El grafo de la figura 2.1 tiene un ciclo formado por la secuencia $(2, 1), (1, 6), (6, 4), (4, 2)$. Sin embargo, el grafo de la figura 2.2 no tiene ningún ciclo.

Dos nodos o vértices están *conectados* si existe un camino que los une.

2.1.2 Tipos de grafos

Se distinguen distintos tipos de grafos por las propiedades que presentan. A continuación definimos los tipos principales.

Un *grafo nulo* es un grafo sin vértices. Un *grafo acíclico* es un grafo que no contiene ciclos. El grafo de la figura 2.2 es acíclico.

Las aristas, al igual que los vértices, pueden tener asociada información que se denomina etiqueta y que puede ser un nombre o un valor, representando una ponderación, peso, coste, distancia etc. según el contexto, y se obtiene así un *grafo etiquetado o valorado*. Un grafo valorado es pues una tripleta $G = \langle N, A, P \rangle$ en la que la pareja (N, A) constituye el conjunto de nodos y aristas, y P es una función que a cada arista de A le asigna un valor de un cierto tipo. Los números asociados a los vértices de la figura 2.1 y de la figura 2.2 son las etiquetas de los vértices y se pueden interpretar como su nombre o identificador. La figura 2.3a muestra un ejemplo de grafo valorado. En este caso las etiquetas de las aristas son números naturales.

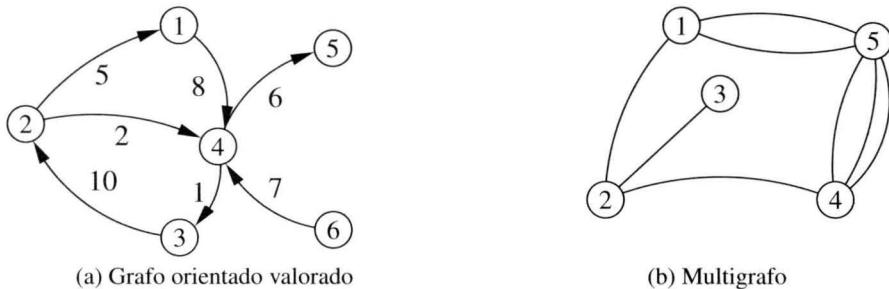


Figura 2.3: Ejemplos de grafos

Se dice que un grafo es *simple* si entre cada par de vértices existe a lo sumo una arista. En caso contrario se trata de un *multigrafo*. La figura 2.3b representa un multigrafo.

Dado un grafo $G = \langle N, A \rangle$ y siendo $N(G)$ y $A(G)$ respectivamente el conjunto de nodos y aristas de G , G' es un *subgrafo* de G si $N(G') \subseteq N(G)$ y si $A(G') \subseteq A(G)$. En el grafo de la figura 2.3a el conjunto de nodos $\{2, 3, 4\}$ y los arcos $\{\langle 2, 4 \rangle, \langle 4, 3 \rangle, \langle 3, 2 \rangle\}$ constituyen un subgrafo.

Se dice que un *grafo* es *conexo* si para cualquier par de vértices distintos existe un camino que los contiene. Un grafo dirigido es *fuertemente conexo* si para cada par de vértices distintos n y m hay un camino de n a m y también hay un camino de m a n . El grafo no dirigido de la figura 2.1 es conexo. El grafo dirigido de la figura 2.3a no es conexo ya que no hay camino para llegar desde el resto de los nodos al nodo 6.

Se denomina *componente conexa* de un grafo a un subgrafo conexo maximal. Maximal se refiere a que dado un subgrafo conexo G' , el grafo no contiene otro subgrafo conexo tal que G' sea un subconjunto de él. El grafo no dirigido de la figura 2.4 tiene dos componentes conexas: una formada por los nodos $\{1, 2, 4, 3\}$ y la otra formada por los nodos $\{5, 6, 7\}$.

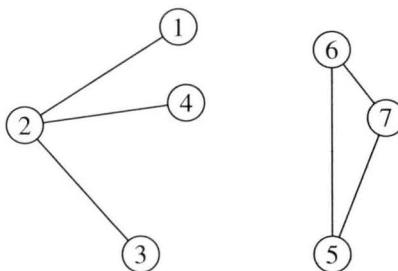


Figura 2.4: Grafo no dirigido con dos componentes conexas

En un grafo dirigido que no es fuertemente conexo, un subgrafo fuertemente conexo maximal recibe el nombre de componente fuertemente conexa del grafo. El grafo di-

rígido de la figura 2.5 tiene dos componentes fuertemente conexas: una formada por los nodos $\{1, 2, 3\}$ y la otra por los nodos $\{4, 5\}$.

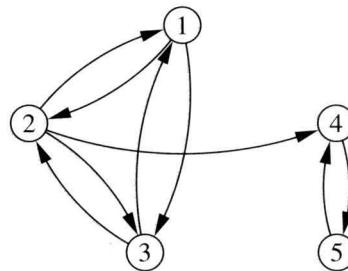


Figura 2.5: Grafo dirigido con dos componentes fuertemente conexas

El máximo número de aristas en un grafo no dirigido de n vértices es $n(n - 1)/2$. Si un grafo no dirigido tiene ese número de aristas se dice que es *completo*. El máximo número de aristas en un grafo dirigido de n vértices es $n(n - 1)$. En ambos casos no se cuentan los bucles.

Un *árbol libre* es un grafo acíclico, conexo y no dirigido. Estos árboles no tienen raíz y los hijos no están ordenados. Para obtener un árbol general hay que seleccionar un nodo como raíz y establecer algún orden entre los hijos de cada nodo.

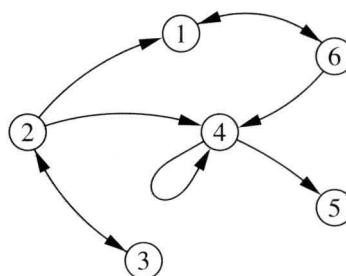
2.1.3 Representación de grafos

Aunque existen diversas opciones para representar un grafo en este texto se presentan las dos más comunes: matriz de adyacencia y listas de adyacencia.

Matriz de adyacencia

Sea $G = \langle N, A \rangle$ un grafo con n vértices. La matriz de adyacencia asociada a G , MA , es una matriz cuadrada de $n \times n$ elementos tal que $MA[i, j] = 1$ si la arista (i, j) (o el arco $\langle i, j \rangle$ si el grafo es dirigido) pertenece a G , y $MA[i, j] = 0$ si no existe tal arista en G . Los valores 1 y 0 se pueden sustituir por valores booleanos (cierto, falso).

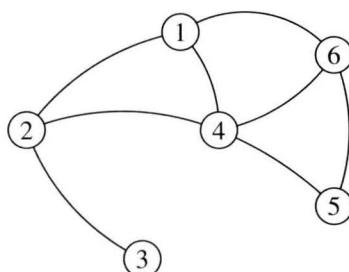
El siguiente grafo dirigido:



se representaría con la matriz de adyacencia:

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| 2 | 1 | 0 | 1 | 1 | 0 | 0 |
| 3 | 0 | 1 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 1 | 1 | 0 |
| 5 | 0 | 0 | 0 | 0 | 0 | 0 |
| 6 | 1 | 0 | 0 | 1 | 0 | 0 |

La matriz de adyacencia de un grafo no dirigido es simétrica, por lo que se puede utilizar y almacenar sólo la matriz triangular superior o inferior. El siguiente grafo no dirigido:



se representaría con la siguiente matriz de adyacencia simétrica:

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| 2 | 1 | 0 | 1 | 1 | 0 | 0 |
| 3 | 0 | 1 | 0 | 0 | 0 | 0 |
| 4 | 1 | 1 | 0 | 0 | 1 | 1 |
| 5 | 0 | 0 | 0 | 1 | 0 | 1 |
| 6 | 1 | 0 | 0 | 1 | 1 | 0 |

Un grafo valorado o etiquetado se puede representar mediante una matriz de adyacencia si en lugar del valor 1 o cierto para representar la existencia de una arista, se utiliza su etiqueta o valor. Si el 0 no es una de las posibles etiquetas, se puede utilizar para representar la no existencia de una arista, en caso contrario habría que utilizar un valor o carácter que no pertenezca al conjunto de posibles valores.

El grafo valorado de la figura 2.3a se representaría:

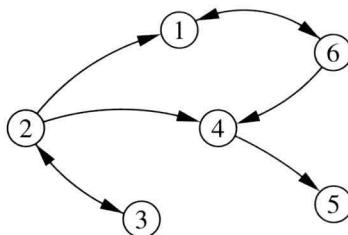
| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|----|---|---|---|---|
| 1 | 0 | 0 | 0 | 8 | 0 | 0 |
| 2 | 5 | 0 | 0 | 2 | 0 | 0 |
| 3 | 0 | 10 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 1 | 0 | 6 | 0 |
| 5 | 0 | 0 | 0 | 0 | 0 | 0 |
| 6 | 0 | 0 | 0 | 7 | 0 | 0 |

Con este tipo de representación, operaciones como determinar cuántas aristas hay en un grafo G de n nodos tienen un coste $O(n^2)$. La representación de un grafo dirigido requiere un espacio en memoria $\Theta(n^2)$, aunque el grafo tenga menos de n^2 arcos.

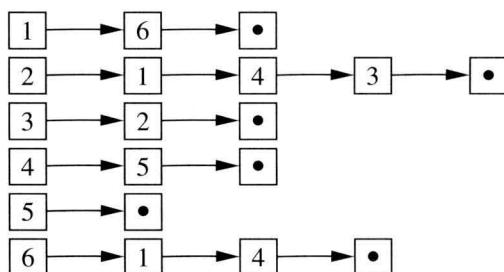
Cuando un grafo es disperso, es decir, tiene pocas aristas y por lo tanto muchos ceros en la matriz de adyacencia, podría ser más conveniente un tipo de representación en la que sólo los nodos y las aristas existentes se representen.

Listas de adyacencia

Las listas de adyacencia son realmente un array de n listas, siendo n el número de nodos, por lo que contiene tantas listas como nodos hay en el grafo. La lista del nodo i tendrá tantos elementos como nodos adyacentes tenga el nodo i . Dentro de cada lista en principio los nodos no están ordenados. El acceso a los nodos adyacentes a uno dado se traduce en acceder y recorrer la lista asociada a dicho nodo. El siguiente grafo dirigido:



se representaría mediante listas de adyacencia de la siguiente manera:



En el caso de grafos etiquetados, las listas deberían incluir un campo adicional para almacenar la etiqueta asociada a cada arista.

Con este tipo de representación, determinar si existe una arista entre el nodo i y el nodo j puede llevar un tiempo $O(n)$, ya que puede haber $O(n)$ vértices en la lista asociada al nodo i . En cuanto al coste en espacio de esta representación, está en $\Theta(n + a)$ siendo n el número de nodos y a el número de aristas.

Funciones de manipulación de grafos

A continuación se muestran los prototipos de las funciones que implementan las operaciones básicas de manipulación de grafos:

Crear grafo: devuelve un grafo vacío.

fun CrearGrafo(): grafo

Añadir arista: añade una arista entre los vértices u y v y le asigna de peso p .

fun AñadirArista (u,v:vértice, p:peso, g:grafo): grafo

Añadir vértice: añade el vértice v al grafo g .

fun AñadirVertice (v:vértice, g:grafo): grafo

Borrar arista: elimina la arista entre los vértices.

fun BorrarArista (v1,v2:vértice, g:grafo): grafo

Borrar vértice: borra el vértice v al grafo g y todas las aristas que partan o lleguen a él.

fun BorrarVertice (v:vértice, g:grafo): grafo

Adyacente?: comprueba si los vértices son adyacentes.

fun Adyacente? (v1,v2:vértice, g:grafo): booleano

Adyacentes: devuelve una lista con los vértices adyacentes a v .

fun Adyacentes (v:vértice, g:grafo): lista

Etiqueta: devuelve la etiqueta o peso asociado a la arista que une los vértices.

fun Etiqueta (v1,v2:vértice, g:grafo): etiqueta

A la hora de calcular los costes se asume lo siguiente:

- CrearGrafo() crea un grafo vacío que corresponderá a una matriz de 0×0 elementos o a un array de listas vacío.
- En la representación con matriz de adyacencia se puede acceder directamente a la fila y columna de un vértice a partir de su identificador. En el caso de las listas de adyacencia se supone que se accede directamente a la posición del vector o array correspondiente al vértice a partir de su identificador. Es decir, no se contempla que sea necesario buscar un vértice en la matriz o en el array de listas, y por lo tanto no se añade ese coste.

Siendo n el número de nodos y a el número de aristas, el coste asociado a estas operaciones en el caso peor dependiendo del tipo de representación queda reflejado en la siguiente tabla:

| | Matriz de adyacencia | Lista de adyacencia |
|---------------|----------------------|---------------------|
| CrearGrafo | $O(1)$ | $O(1)$ |
| AñadirArista | $O(1)$ | $O(1)$ |
| AñadirVertice | $O(n)$ | $O(1)$ |
| BorrarArista | $O(1)$ | $O(n)$ |
| BorrarVertice | $O(n)$ | $O(n+a)$ |
| Adyacente? | $O(1)$ | $O(n)$ |
| Adyacentes | $O(n)$ | $O(1)$ |
| Etiqueta | $O(1)$ | $O(n)$ |

Si el grafo tiene pocas aristas las listas de adyacencia resultan menos costosas en espacio. Sin embargo, cuando a se acerca a n^2 (máximo número de nodos posibles) el coste en espacio es del mismo orden. En este caso, al decidir entre las dos representaciones se podría tener en cuenta que la matriz de adyacencia es más sencilla de tratar ya que no utiliza punteros.

En cuanto al coste en tiempo, dependiendo de qué operaciones se necesiten será más apropiada una representación que otra. Por ejemplo, si el algoritmo necesita recorrer todas las aristas de un grafo, con la matriz de adyacencia tendrá un coste $O(n^2)$, mientras que con las listas de adyacencia tendrá un coste $O(n+a)$, que resultará muy ventajoso si el grafo tiene pocas aristas.

En general, para grafos dispersos, con pocas aristas, las listas de adyacencia son el tipo de representación más eficiente; para grafos densos, con muchas aristas, la matriz de adyacencia resulta más apropiada.

2.1.4 Recorrido de grafos

La resolución de muchos de los problemas que se pueden formular en términos de grafos en ocasiones requiere visitar todos los nodos y/o aristas. En este apartado vamos a ver dos tipos de recorrido de grafos que se pueden aplicar tanto a grafos dirigidos como a grafos no dirigidos.

Recorrido en profundidad

En la bibliografía también se conoce a este recorrido como búsqueda primero en profundidad (*depth-first search*). Inicialmente se marcan todos los nodos como no visitados y se selecciona un nodo u como punto de partida. A continuación, se marca como visitado y se accede a un nodo no visitado v adyacente al nodo u . Se procede recursivamente con

el nodo v . Al volver de la llamada o llamadas recursivas, si hay algún nodo adyacente que no se ha visitado, se toma como punto de partida y se vuelve a ejecutar el procedimiento recursivo. El recorrido termina cuando todos los nodos están marcados como visitados. Este recorrido sería equivalente al recorrido en preorden de un árbol.

Dado un nodo inicial, la siguiente función recursiva se encargaría del recorrido en profundidad recursivo:

```
fun RecProfundidadRecursivo(v: nodo, visitado: Vector)
    var
        w: nodo
    fvar
        visitado[v] ← cierto
    para cada w adyacente a v hacer
        si ¬ visitado[w] entonces
            RecProfundidadRecursivo(w, visitado)
        fsi
    fpara
ffun
```

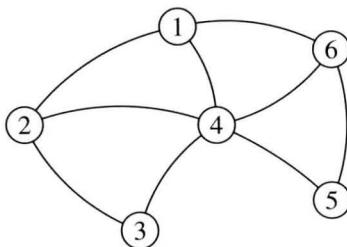
Para asegurar que se visitan todos los nodos de un grafo, tanto si es conexo como si no lo es, la función anterior debería ser llamada desde la siguiente:

```
tipo Vector = matriz[0..n] de booleano
fun RecorridoProfundidad( $G = \langle N, A \rangle$ : grafo)
    var
        visitado: Vector
        v: nodo
    fvar
        para cada v  $\in N$  hacer
            visitado[v] ← falso
        fpara
        para cada v  $\in N$  hacer
            si ¬ visitado[v] entonces
                RecProfundidadRecursivo(v, visitado)
            fsi
        fpara
ffun
```

Este recorrido se puede realizar tanto en grafos dirigidos como no dirigidos; la única diferencia es lo que se entiende por adyacente en uno u otro tipo de grafo. Si el grafo es dirigido, el nodo w es adyacente al nodo u si existe el arco $\langle u, w \rangle$. Si no existe el arco $\langle w, u \rangle$ entonces el nodo u no es adyacente al nodo w .

En cuanto al coste del recorrido en profundidad, si consideramos que n es el número de nodos y a es el número de aristas de un grafo, en su recorrido en profundidad sólo se hace una llamada a *RecProfundidadRecursivo* por cada nodo no visitado, por lo que *RecProfundidadRecursivo* se ejecuta n veces. La obtención de los adyacentes depende de la forma de representación del grafo. Si la representación es mediante una matriz de adyacencia, los adyacentes a un nodo se obtienen recorriendo una fila de la matriz $n \times n$, por lo que el coste de la búsqueda en profundidad es de $O(n^2)$. Si la representación es mediante listas de adyacencia, para obtener los adyacentes a un nodo habrá que recorrer su lista de adyacentes. Sabemos que la suma de las longitudes de las listas de adyacencia que representan un grafo es a , así el tiempo estaría en $O(n + a)$.

En la figura 2.6 se puede ver un grafo no dirigido y la secuencia de llamadas a *RecProfundidadRecursivo* cuando se toma como nodo inicial del recorrido el nodo 1 y se visitan los adyacentes a un nodo dado por orden numérico.



(a) Grafo no dirigido

RecProfundidadRecursivo(1)
RecProfundidadRecursivo(2)
RecProfundidadRecursivo(3)
RecProfundidadRecursivo(4)
RecProfundidadRecursivo(5)
RecProfundidadRecursivo(6)

(b) Secuencia de llamadas del recorrido en profundidad partiendo del nodo 1

Figura 2.6: Ejemplo de recorrido en profundidad de un grafo

También se puede implementar el recorrido en profundidad en forma iterativa. Para ello, necesitamos una estructura de datos Pila (“Last Input First Output” LIFO) con las operaciones típicas del Tipo Abstracto de Datos (TAD) pila: PilaVacia, Apilar, Desapilar y Cima.

```
fun RecProfundidadIterativo(v: nodo, visitado: Vector)
```

```
var
```

```
w: nodo
```

```
P: TPila
```

```
fvar
```

```
P ← PilaVacía
```

```
visitado[v] ← cierto
```

```
Apilar(v,P)
```

```
mientras ¬ vacia(P) hacer
```

```
para cada w adyacente a Cima(P) hacer
```

```
si ¬ visitado[w] entonces
```

```

visitado[w] ← cierto
Apilar(w,P)
fsi
fpara
Desapilar(P)
fmientras
ffun

```

Al igual que en la versión recursiva, para asegurar que se recorren todos los nodos del grafo, tanto si es conexo como si no lo es, haría falta llamarlo desde:

```

tipo Vector = matriz[0..n] de booleano
fun RecorridoProfundidad( $G = \langle N, A \rangle$ : grafo)
    var
        visitado: Vector
        v: nodo
    fvar
    para cada v ∈ N hacer
        visitado[v] ← falso
    fpara
    para cada v ∈ N hacer
        si ¬ visitado[v] entonces
            RecProfundidadIterativo(v, visitado)
        fsi
    fpara
ffun

```

El cálculo del coste es similar al de la versión recursiva.

El recorrido en profundidad también permite numerar los nodos del grafo de acuerdo al orden en que se visitan. Con esta nueva funcionalidad, el algoritmo de recorrido en profundidad recursivo quedaría:

```

fun RecProfundidadRecursivoNum(v: nodo, num: natural, visitado: Vector, numOrden: VectorNat)
    var
        w: nodo
    fvar
        visitado[v] ← cierto
        num ← num + 1
        numOrden[v] ← num
    para cada w adyacente a v hacer

```

si \neg visitado[w] **entonces**

 RecProfundidadRecursivoNum(w, num, visitado, numOrden)

fsi

fpara

ffun

Este algoritmo se llamaría desde el siguiente:

tipo Vector = matriz[0..n] de booleano

tipo VectorNat = matriz[0..n] de natural

fun RecorridoProfundidadNum($G = \langle N, A \rangle$: grafo)

var

 visitado: Vector

 numOrden: VectorNat

fvar

para cada $v \in N$ **hacer**

 visitado[v] \leftarrow falso

fpara

 num \leftarrow 0

para cada $v \in N$ **hacer**

si \neg visitado[v] **entonces**

 RecProfundidadRecursivoNum(v, num, visitado, numOrden)

fsi

fpara

ffun

En el grafo ejemplo de la figura 2.6a el orden de visita de los nodos coincidiría con el orden en el que son llamados por la función *RecProfundidadRecursivo()* que aparece en la figura 2.6b: el nodo 1 se recorrería el primero, el nodo 2 el segundo, el 3 el tercero, el 4 el cuarto, el 5 el quinto y el 6 en sexto lugar.

Recorrido en amplitud o en anchura

En la bibliografía también se conoce a este recorrido como búsqueda primero en anchura (*breadth-first search*). Inicialmente se marcan todos los nodos como no visitados y se selecciona un nodo u como punto de partida. A continuación, se marca como visitado y se visitan todos los nodos no visitados adyacentes al nodo u . A continuación se procede de manera similar con los adyacentes a cada uno de los nodos recién visitados. Se puede considerar que es un recorrido por niveles, primero se accede a los nodos que están a una arista de distancia del nodo inicial del recorrido, después a los que están a dos aristas de distancia, y así sucesivamente hasta que se visitan todos los nodos accesibles desde el inicial.

El recorrido en anchura se emplea cuando hay que realizar una exploración parcial de un grafo infinito, o potencialmente muy grande, y también para calcular el camino más corto desde un punto del grafo hasta otro.

El recorrido en anchura no es de naturaleza recursiva, se necesita una estructura de datos lista de tipo “First Input First Output” (FIFO) que corresponde a una Cola. Las operaciones típicas del TAD cola que se van a usar son: ColaVacia, EnColar, DesenColar y Primero.

Dado un nodo inicial, la siguiente función se encargaría del recorrido en anchura:

```
fun RecAnchura(v: nodo, visitado: Vector)
    var
        u,w: nodo
        Q: TCola
    fvar
        Q ← ColaVacia
        visitado[v] ← cierto
        Encolar(v,Q)
    mientras ¬ vacia(Q) hacer
        u ← Primero(Q)
        Desencolar(u,Q)
        para cada w adyacente a u hacer
            si ¬ visitado[w] entonces
                visitado[w] ← cierto
                Encolar(w,Q)
        fsi
    fpara
    mientras
ffun
```

Al igual que en el recorrido en profundidad, para asegurar que se recorren todos los nodos del grafo, tanto si es conexo como si no lo es, haría falta llamarlo desde la siguiente función:

```
tipo Vector = matriz[0..n] de booleano
fun RecorridoAnchura( $G = \langle N, A \rangle$ : grafo)
    var
        visitado: Vector
        v: nodo
    fvar
        para cada v ∈ N hacer
            visitado[v] ← falso
```

```

fpara
para cada v  $\in$  N hacer
    si  $\neg$  visitado[v] entonces
        RecAnchura(v, visitado)
    fsi

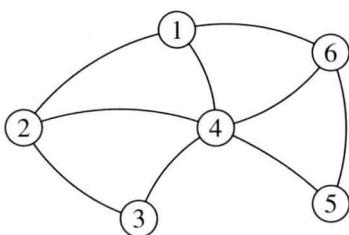
```

fpara

ffun

El razonamiento y cálculo del coste del recorrido en anchura es similar al del recorrido en profundidad: $O(n^2)$ cuando se representa el grafo con una matriz de adyacencia y $O(n + a)$ cuando se representa mediante listas de adyacencia.

En la figura 2.7 se puede ver un grafo no dirigido y el contenido de la cola cuando se toma como nodo inicial del recorrido el nodo 1, y se visitan los adyacentes a un nodo dado por orden numérico.



(a) Grafo no dirigido

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

(b) Contenido de la cola del recorrido en anchura partiendo del nodo 1

Figura 2.7: Ejemplo de recorrido en anchura de un grafo

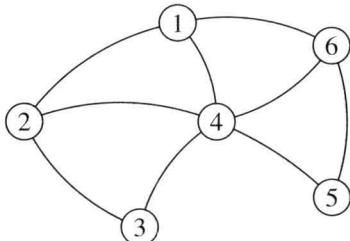
Este recorrido también se puede realizar tanto en grafos dirigidos como no dirigidos. Al igual que en recorrido en profundidad la única diferencia es lo que se entiende por adyacente en uno u otro tipo de grafo.

2.1.5 Árboles de recubrimiento

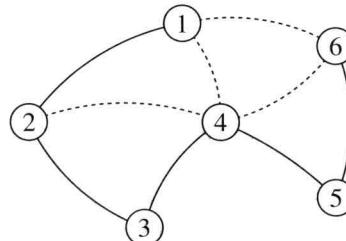
Los recorridos en profundidad o en anchura de un grafo conexo le asocian un árbol de recubrimiento. Así, las aristas del grafo se dividen en dos conjuntos: las que pertenecen al árbol y las que no. Las aristas que no forman parte del árbol son las que no se usan en el recorrido del grafo. El nodo inicial del recorrido es la raíz del árbol.

La figura 2.8a muestra un grafo no dirigido y la figura 2.8b el árbol de recubrimiento asociado por el recorrido en profundidad partiendo del nodo 1. Las aristas en línea discontinua representan las que no se han utilizado en el recorrido debido a que conducían a nodos que ya estaban visitados. Las aristas que sí se han utilizado en el recorrido, en el sentido de que conducían a nodos sin visitar, son las que forman parte del árbol de

recubrimiento. Se observa como se han eliminado los ciclos del grafo original. Si G es el grafo y AR es un árbol de recubrimiento asociado, se da que una arista de G que no esté en AR une necesariamente un nodo con alguno de sus antecesores.



(a) Grafo no dirigido



(b) Aristas del árbol de recubrimiento asociado

Figura 2.8: *Ejemplo de árbol de recubrimiento a partir de un recorrido en profundidad*

Si el grafo no es conexo, el recorrido en profundidad le asocia un bosque de árboles, uno por cada componente conexa del árbol. Veamos el ejemplo de la figura 2.9. En la subfigura 2.9a se puede ver un grafo dirigido con 2 componentes conexas: una formada por los nodos $\{1,2,3,4,5\}$ y la otra por los nodos $\{6,7\}$. Si se hace un recorrido en profundidad tomando el nodo 1 como origen y visitando los adyacentes a un nodo dado por orden numérico, se observa que se obtiene un bosque de recubrimiento formado por 2 árboles, uno por cada componente conexa. Para este ejemplo, la secuencia de llamadas a la función de recorrido en profundidad recursiva sería la siguiente:

```

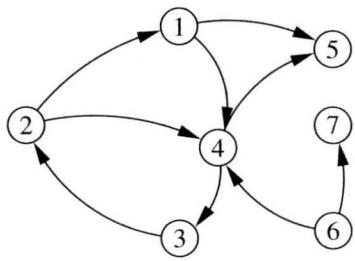
RecProfundidadRecursivo(1)
  RecProfundidadRecursivo(4)
    RecProfundidadRecursivo(3)
      RecProfundidadRecursivo(2)
      RecProfundidadRecursivo(5)
    RecProfundidadRecursivo(6)
    RecProfundidadRecursivo(7)

```

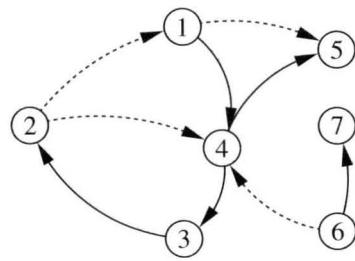
La búsqueda en profundidad puede dar lugar a diferentes árboles o bosques de recubrimiento según el orden en el que se examinen los adyacentes.

En un grafo dirigido las aristas del grafo que no están en el árbol o bosque asociado pueden ser de 3 tipos:

- Las que unen un nodo con uno de sus antecesores. Véanse los arcos $\langle 2,1 \rangle$ y $\langle 2,4 \rangle$ en la figura 2.9b.
- Las que unen un nodo con uno de sus descendientes o sucesores. Véase la arista $\langle 1,5 \rangle$ en la figura 2.9b.



(a) Grafo dirigido no conexo



(b) Bosque de recubrimiento asociado

Figura 2.9: Ejemplo de bosque de recubrimiento a partir de un recorrido en profundidad

- Las que unen un nodo con otro que no es antecesor ni descendiente suyo. Véase la arista $\langle 6,4 \rangle$ en la figura 2.9b.

En el caso de grafos dirigidos las aristas solo pueden unir un nodo con un antecesor o con un descendiente.

2.1.6 Puntos de articulación

Dado un grafo conexo, un nodo u es un punto de articulación si al eliminar u y todas las aristas que inciden en él el grafo deja de ser conexo. Por ejemplo, en el grafo de la figura 2.10, el nodo 1 es un punto de articulación ya que si se elimina junto con sus aristas el grafo dejaría de ser conexo y pasaría a tener dos componentes conexas: $\{2, 4, 3\}$ y $\{5, 6, 7\}$. El nodo 5 también es un punto de articulación ya que al eliminarlo a él y a sus aristas el grafo quedaría con las componentes conexas: $\{1, 2, 4, 3\}$ y $\{6, 7\}$. Lo mismo ocurre con el nodo 2, quedando dos componentes conexas: $\{3\}$ y $\{1, 4, 5, 6, 7\}$. Sin embargo, los nodos 3, 4, 6 y 7 no son puntos de articulación.

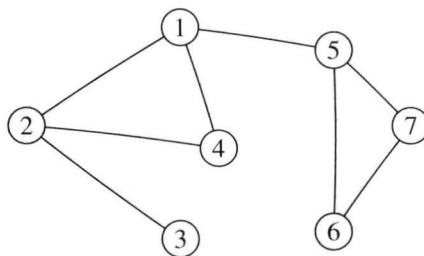


Figura 2.10: Ejemplo de grafo con puntos de articulación

Un grafo conexo sin puntos de articulación se llama *biconexo*. Encontrar los puntos de articulación de un grafo es una tarea básica en problemas de conectividad de grafos. Supongamos que tenemos una red de comunicaciones representada como un grafo, en el que los nodos son puntos que se quieren mantener en conexión y las aristas son las

líneas de conexión. Si el grafo es biconexo, la red de comunicaciones seguirá en funcionamiento aunque falle un nodo. Un grafo tiene *conectividad k* si la eliminación de $k - 1$ nodos cualesquiera no lo desconecta. Un grafo biconexo tiene al menos conectividad 2. Cuanto mayor sea el valor de k , es decir su conectividad, será más resistente a posibles fallos en los nodos.

La búsqueda de los puntos de articulación de un grafo se basa en el recorrido en profundidad. En el caso de que no se encuentren se concluirá que el grafo es biconexo.

Los pasos para hallar los puntos de articulación de un grafo son los siguientes:

1. Realizar un recorrido en profundidad del grafo numerando los nodos según avanza el recorrido (ver como numerar los nodos en 2.1.4). Como resultado tendremos el grafo inicial G , un árbol de recubrimiento asociado, AR , junto con el número de orden que le asigna el recorrido a cada nodo. En la figura 2.11 se puede ver un grafo, el árbol de recubrimiento asociado y la numeración de los nodos ($numOrden[]$).
2. Recorriendo el árbol de recubrimiento AR en postorden¹ se calcula para cada nodo visitado v el valor $bajo[v]$. El valor de $bajo[v]$ representa el nodo más alto al que podemos llegar desde v en AR descendiendo por 0 o más aristas del árbol y ascendiendo como máximo por una arista que no pertenece al árbol. En la figura 2.11b las aristas que no pertenecen al árbol están en línea discontinua. Hay que tener en cuenta que si un nodo no tiene hijos en AR no puede ser un punto de articulación, ya que al eliminarlo los nodos restantes seguirían conectados mediante las aristas que quedan. Para cada nodo v visitado, $bajo[v]$ se calcula como el valor mínimo de:
 - (a) $numOrden[v]$,
 - (b) $numOrden[w]$ para cualquier w tal que haya una arista de retroceso (v, w) en G que no esté en AR , y
 - (c) $bajo[x]$ para cualquier hijo x de v .

En la figura 2.12 se han calculado los valores de $bajo[]$ que son los que aparecen entre paréntesis. El recorrido en postorden analizaría los nodos en el siguiente orden: primero el nodo 3, luego el 4, 2, 7, 6, 5 y la raíz 1. El valor $bajo[3]$ es 3 ya que no se puede ir a un punto más alto siguiendo una arista que no esté en el grafo, ni tiene descendientes, por lo que el valor de $bajo[3]$ es $numOrden[3]$. El valor $bajo[4]$ es 1 ya que se puede llegar al nodo 1 ascendiendo por una línea discontinua. El valor $bajo[2]$ es 1 ya que el nodo 4 es descendiente suyo y a través de él puede llegar al 1 que tiene valor menor. El valor $bajo[7]$ es 5 ya que se puede llegar al nodo 5 ascendiendo por una línea discontinua. El valor $bajo[6]$ es 5 ya que el nodo 7 es descendiente suyo y a través de él puede llegar al 5 que tiene valor

¹Se recuerda que el recorrido en postorden de un árbol recorre primero el subárbol izquierdo, después el subárbol derecho y, por último, la raíz.

menor. El valor $bajo[5]$ es 5 el $numOrden[5]$ y, finalmente, el valor de $bajo[1]$ es 1 el $numOrden[1]$.

3. Se calculan los puntos de articulación de la siguiente manera:

- (a) La raíz de A es un punto de articulación si tiene más de un hijo.
- (b) Cualquier otro nodo v es un punto de articulación si tiene un hijo w tal que $bajo[w] \geq numOrden[v]$. En este caso al eliminar v y sus aristas se desconectaría w . El caso contrario, $bajo[w] < numOrden[v]$, indica que hay un camino para moverse desde el nodo w a un antecesor del nodo v y por lo tanto la eliminación de v no desconectaría a w del resto del grafo.

En el ejemplo de la figura 2.12 los nodos 1, 2, y 5 son puntos de articulación.

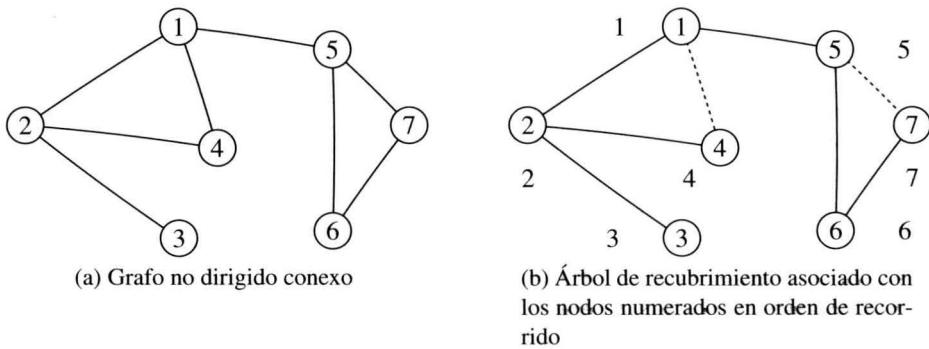


Figura 2.11: Ejemplo de grafo con puntos de articulación y su árbol de recubrimiento asociado

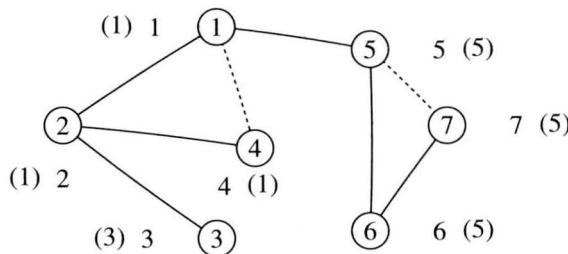


Figura 2.12: Ejemplo de grafo con los valores de $numOrden[]$ y $bajo[]$ calculados

2.1.7 Ordenación topológica de un grafo dirigido acíclico

Un grafo dirigido acíclico (gda) es un grafo dirigido que no tiene ciclos. Los gda permiten modelar de manera natural ciertos problemas cuya resolución requiere la realización de una serie de tareas en un orden específico. En un gda la existencia del arco $\langle u, v \rangle$ indica que el nodo u precede al nodo v en una ordenación lineal.

Por ejemplo, los requisitos de un plan de estudios de una carrera se pueden modelar con un gda. Si los requisitos se refieren a asignaturas, cada asignatura sería un nodo y los arcos indicarían cuál hay que cursar antes. Si los requisitos se refieren a cursos sería igual pero en relación a cursos.

Algunos problemas relacionados con la planificación de tareas también se pueden modelar con un gda. Por ejemplo, en el caso de que haya que realizar una serie de tareas de forma que una no puede empezar hasta que otra u otras acaben.

Los gda también pueden representar expresiones aritméticas con subexpresiones repetidas. La figura 2.13 representa la estructura de la siguiente expresión mediante un gda:

$$\frac{(a - b)(c + d)}{(a + b)(c + d)}$$

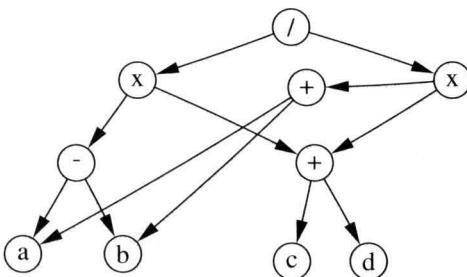


Figura 2.13: Un grafo dirigido acíclico

La clasificación topológica asigna un orden lineal a los nodos de un gda. En el caso de una expresión aritmética indicaría el orden en que deben calcularse las subexpresiones para obtener el resultado.

Para determinar si un grafo dirigido es acíclico se puede utilizar un recorrido en profundidad. También se puede utilizar el recorrido en profundidad para determinar la ordenación topológica de un gda, añadiendo una instrucción de escritura a la función recursiva de recorrido en profundidad:

```

fun RecProfundidadRecursivoOrdenTopologico(v: nodo, visitado: Vector)
  var
    w: nodo
  fvar
    visitado[v] ← cierto
    para cada w adyacente a v hacer
      si ¬ visitado[w] entonces
        RecProfundidadRecursivoOrdenTopologico(w, visitado)
    fsi
  
```

```

fpara
  escribir(v)
ffun

```

Para asegurar que se visitan todos los nodos del grafo se llamaría a la función anterior desde la función *RecorridoProfundidad* vista en el apartado 2.1.4.

El recorrido deberá comenzar por un nodo que no tiene ningún predecesor. Esta función imprime los nodos en orden topológico inverso. Veamos otro ejemplo de gda en la figura 2.14. En este caso el gda representa una planificación de tareas. Si lo recorriéramos siguiendo la función anterior empezando en el nodo T1 escribiría: T6 T5 T3 T1 T4 T2. Al invertir la lista anterior tendríamos un orden que garantizaría la ejecución de todas las tareas teniendo en cuenta las precedencias.

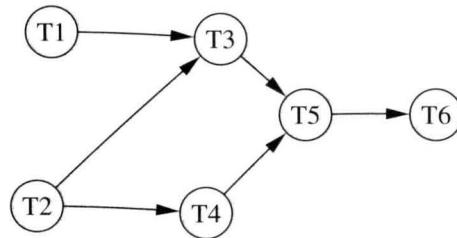


Figura 2.14: Un grafo dirigido acíclico

En el caso del ejemplo de la figura 2.13, la función escribiría: $a b - c d + x a b + c d + x /$. Se puede observar que evaluando las subexpresiones en ese orden se puede calcular el resultado de la expresión. Nótese que en el caso de las expresiones aritméticas, los nodos terminales del grafo no representan el objetivo, sino las subexpresiones más elementales.

2.1.8 Camino más corto desde la raíz a cualquier otro nodo

Dado un grafo G , se define la distancia más corta entre los vértices u y v del grafo G como el mínimo número de aristas en cualquier camino entre u y v en G . Sea $G = \langle N, A \rangle$ un grafo y sea $ARA = \langle N, A' \rangle$ el árbol de recubrimiento asociado al recorrido en anchura de G . Para cada vértice w , la longitud del camino desde la raíz hasta w en ARA coincide con la longitud del camino más corto desde la raíz hasta w en G .

Por ejemplo, el problema de atravesar un laberinto, considerando el punto de entrada del laberinto como la raíz, se puede resolver con un recorrido en anchura. También cuando partimos de un valor numérico inicial y queremos llegar a obtener un valor n aplicando una serie de operaciones aritméticas. Este problema es básicamente una búsqueda en un grafo dirigido infinito, que también se puede resolver con un recorrido en anchura. Supongamos que el valor inicial es 2 y que utilizando dos operaciones, multiplicación por 3 y división entera entre 2, queremos llegar al valor 4 en el menor número de pasos.

En la figura 2.15 tenemos la parte inicial del grafo que describe el conjunto de valores de las operaciones anteriores. Para cada nodo, la arista superior se refiere a la multiplicación por 3 y la inferior a la división entre 2.

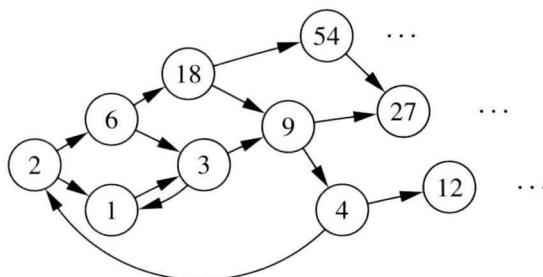


Figura 2.15: Un grafo dirigido infinito

Si utilizamos el recorrido en anchura aplicando primero la operación división y luego la multiplicación, el valor 4 lo obtendríamos con la secuencia: $2 \div 2 \times 3 \times 3 \div 2$. Si utilizáramos un recorrido en profundidad podríamos seguir una rama que no llegara nunca al valor 4.

2.1.9 Otros algoritmos sobre grafos

Existen otros algoritmos con grafos que se podrían incorporar a este capítulo. Sin embargo, dado que el libro también trata esquemas algorítmicos, y dichos algoritmos con grafos son representativos de algunos de los esquemas que se van a estudiar, se deja su descripción al capítulo correspondiente de algoritmos. Por ejemplo, los algoritmos de cálculo de un árbol de recubrimiento mínimo y de caminos mínimos se estudiarán en el capítulo dedicado a los algoritmos voraces (ver capítulo 3). Lo mismo ocurrirá con otros algoritmos que exploran espacios de búsqueda que se modelan mediante un grafo.

2.2 Montículos

Los montículos son un tipo especial de árbol binario que se implementan sobre vectores con las siguientes propiedades:

- Es un árbol balanceado y completo (los nodos internos tienen siempre dos hijos) con la posible excepción de un único nodo cuando el número de elementos es impar.
- Cada nodo contiene un valor mayor o igual que el de sus nodos hijos (montículo de máximos), o menor o igual (montículo de mínimos).

A la segunda de estas propiedades se le denomina *propiedad de montículo* y permite tener en la cima del montículo (el nodo raíz) el elemento mayor (o menor si se trata

de un montículo de mínimos), siendo ésta la utilidad fundamental del montículo. Los nodos de profundidad k están situados en las posiciones 2^k y siguientes del vector hasta la $2^{k+1} - 1$.

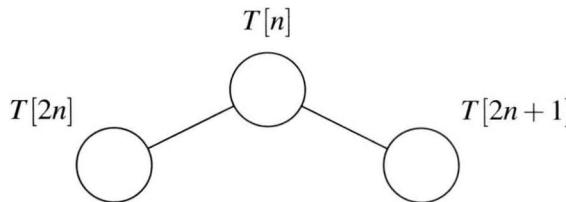


Figura 2.16: *Estructura básica de un montículo*

Como hemos comentado, la implementación no se realiza utilizando árboles, sino vectores, cuyo recorrido es más eficiente. El vector $T[1..n]$ implementa un montículo cuando:

- El nodo raíz es el elemento $T[1]$ del vector y contiene el mayor (o menor) de los elementos del montículo.
- Los nodos hijos del elemento $T[i]$ que son respectivamente $T[2i]$ y $T[2i + 1]$ cumplen que $T[i] \geq T[2i]$ y $T[i] \geq T[2i + 1]$ para el caso de los montículos de máximos.

Para acceder al padre del nodo $T[i]$ se accede al elemento $T[i \text{ div } 2]$. Asumiremos sin pérdida de generalidad que los montículos de ejemplo del resto del capítulo implementan montículos de máximos.

Al cumplir la *propiedad de montículo* que permite tener en la cima el elemento más prometedor, la principal utilidad es esta estructura de datos es la de proporcionar un método eficiente de implementar colas de prioridad. El orden interno de los elementos no es relevante, pero sí lo es la eficiencia en recuperar la propiedad de montículo ante operaciones básicas. El montículo fue desarrollado por Floyd como estructura de datos de apoyo a un método de ordenación denominado *Heapsort* (ordenación por montículo), que se menciona en el capítulo 4.

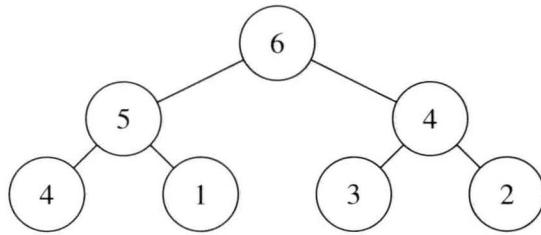
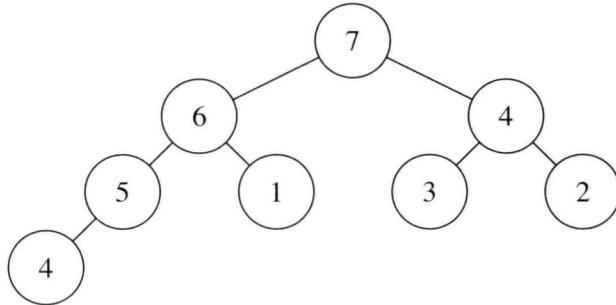
Podemos ver estas propiedades con un ejemplo. El vector

$$m = [6, 5, 4, 4, 1, 3, 2]$$

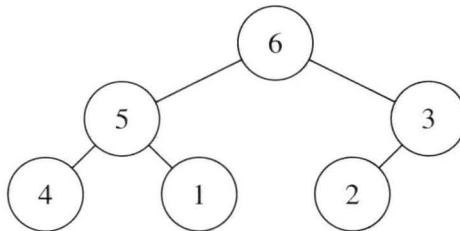
implementa el montículo de a figura 2.17, representado en forma de árbol.

Al insertar un elemento, por ejemplo añadir el valor 7, el montículo queda como muestra la figura 2.18.

Como se puede comprobar, los montículos no son vectores ordenados, si bien el mayor elemento ocupa la cima del mismo. De esta manera el objetivo de esta estructura de

Figura 2.17: Un montículo de máximos M Figura 2.18: M tras añadir 7

datos es tener en todo momento en la cima del mismo al mayor de sus elementos. La particularidad de los montículos es que es una estructura de datos que permite de manera muy eficiente insertar y borrar un elemento y restaurar la propiedad de montículo.

Figura 2.19: M tras eliminar la cima

En la figura 2.19 tenemos el resultado de eliminar la cima del montículo. El resultado es que este se recomponen y recupera la propiedad de montículo perdida en la figura 2.18. Una estructura de montículo tiene aplicaciones importantes en otros algoritmos conocidos:

- Algoritmos de ordenación como *Heapsort*, uno de los algoritmos más relevantes con un coste eficiente en el caso peor.
- Algoritmos de selección: la búsqueda de mínimos, máximos, medianas o el k-ésimo mayor elemento pueden realizarse en tiempo lineal mediante el uso de montículos.
- Algoritmos basados en grafos: usando montículos como estructuras internas de almacenamiento de nodos o aristas, la complejidad puede reducirse en un orden polinómico. Ejemplos de este tipo de algoritmos son la creación de árboles de recubrimiento mínimo de Prim o el problema del camino mas corto de Dijkstra.

2.2.1 Implementación y operaciones sobre elementos del montículo

La estructura de datos para implementar el montículo consta de un registro con tres elementos:

- Un vector $T[1..n]$.
- Un contador c para el numero de elementos del montículo.
- Un valor para el tamaño máximo del montículo.

que se puede definir como sigue:

```
registro monticulo
  T: vector [1..n] de entero;
  c: natural;
  MAX: natural;
```

fregistro

El montículo tiene las operaciones básicas siguientes:

CreaMonticuloVacio: devuelve un montículo vacío.

MonticuloVacio?: devuelve **cierto** si el montículo está vacío.

Flotar: reubica el elemento i -esimo del vector en caso de que este sea mayor que el padre.

Hundir: reubica el elemento i -esimo del vector en caso de que éste sea menor que alguno de sus hijos. En tal caso, intercambia su valor por el del mayor de sus hijos.

Insertar: inserta un elemento en el montículo y lo *flota* hasta restaurar la propiedad de montículo.

Primero: devuelve la cima del montículo sin modificarlo.

ObtenerCima: devuelve la cima del montículo, la elimina del mismo y recompone la propiedad de montículo.

Describimos las funciones a continuación:

Función *CreaMontículoVacio*

La función devuelve un montículo vacío con el contador iniciado a 0 elementos.

```
fun CreaMonticuloVacio(m: monticulo)
    m.T ← null
    m.c ← 0
    m.MAX ← n
ffun
```

Función *MontículoVacio?*

La función devuelve un montículo vacío con el contador iniciado a 0 elementos.

```
fun MonticuloVacio?(m: monticulo)
    si (m.c = 0) entonces dev cierto sino dev falso
ffun
```

Función *Flotar*

La función flotar reubica el elemento *i-esimo* del vector *T* del montículo en caso de que éste sea mayor que el padre, hasta que esté correctamente situado en el montículo y se haya restablecido la *propiedad de montículo*. El proceso de flotar se utiliza para la inserción de un elemento nuevo en el montículo.

```
fun flotar(T: vector, i:natural)
    var
        padre:natural
```

fvar

```
padre ← i div 2
```

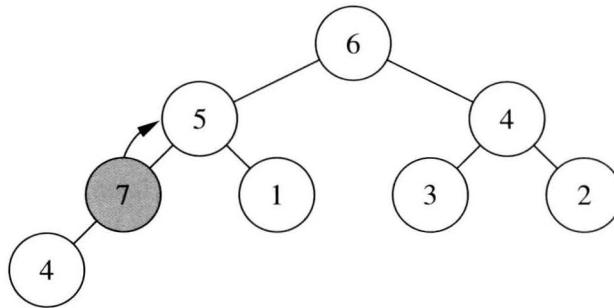
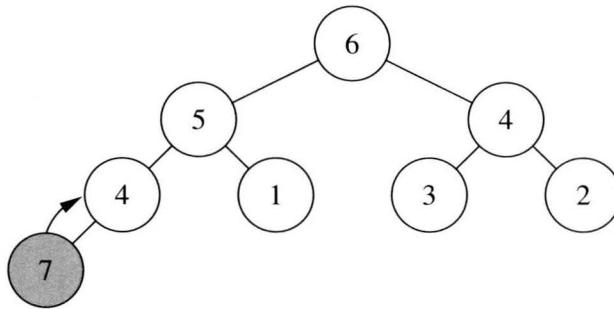
```
mientras (i>1) ∧ (T[padre] < T[i]) hacer
```

```
    intercambiar (T[i],T[padre])
```

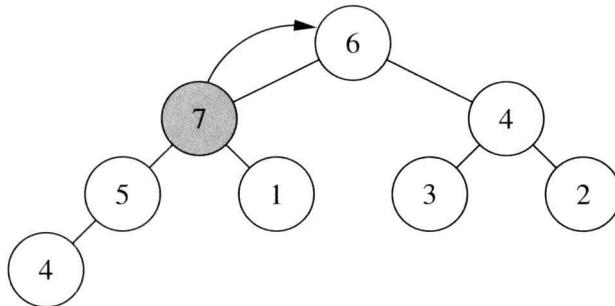
```
    i ← i div 2
```

fmientras**ffun**

Supongamos que tenemos el siguiente caso, en el que aplicamos la función *flotar* al montículo siguiente (en oscuro se señala aquel nodo que no cumple la propiedad de montículo):

**Función Hundir**

La función *Hundir* reubica el elemento i -ésimo del vector T del montículo m en caso de que este sea menor que alguno de sus hijos. En tal caso, intercambia su valor por el del mayor de sus hijos.

Figura 2.20: *Operación flotar*

El algoritmo de la función *Hundir* es el siguiente:

```

fun Hundir(T:vector, i:natural)
  var
    hi,hd,p:natural
  fvar
    hi  $\leftarrow$  2*i
    hd  $\leftarrow$  2*i+1
    p  $\leftarrow$  i
  repetir
    si (hi  $\leq$  m.MAX)  $\wedge$  (T[hd] > T[i]) entonces
      i  $\leftarrow$  hd
    fsi
    si (hi  $\leq$  m.MAX)  $\wedge$  (T[hi] > T[i]) entonces
      i  $\leftarrow$  hi
    fsi
    intercambiar (T[p],T[i])
    i  $\leftarrow$  i div 2
  hasta p=i;
ffun
  
```

En el montículo de la figura 2.21, el elemento de la cima no cumple la propiedad de montículo. Realizamos sobre este elemento la operación *Hundir* y tras ella, los elementos del vector recuperan la *propiedad de montículo*.

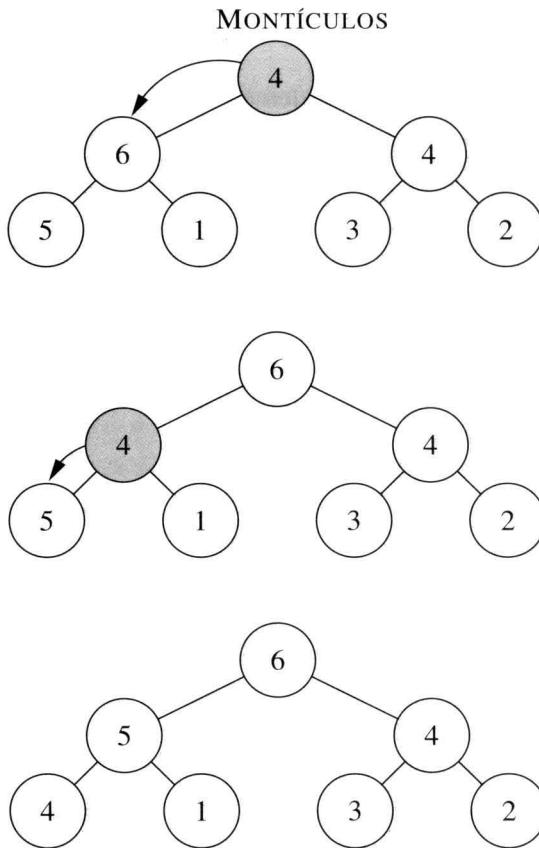


Figura 2.21: *Operación hundir*

Función Insertar

La manera más intuitiva de inserción de un elemento en un montículo se realiza incorporando el elemento al final del mismo y aplicando la operación *Flotar*.

```

fun Insertar(e:elemento; m: montículo): monticulo
    si m.c = m.MAX entonces
        error(MonticuloLleno)
    sino
        m.c ← m.c + 1
        m.T[m.c] ← e
        Flotar(m.T,m.c)
    fsi
ffun

```

El ejemplo anterior es el caso de añadir el elemento 7 al montículo. En un principio el montículo consta de 7 elementos, con lo que se inserta en la posición 8, con $m.T[8] = 7$ y $m.c = 8$, y a continuación se invoca a $flotar(m, 8)$.

Función *Primero*

Esta función devuelve el valor que hay en la cima del montículo sin eliminarla.

```
fun Primero(m: montículo): elemento
    si m.c = 0 entonces dev error
    sino
        dev m.T[1]
    fsi
ffun
```

Se trata de una función de coste = (1).

Función *ObtenerCima*

El borrado de la cima del montículo devuelve el valor de esta y elimina la cima de la primera posición del montículo. Para recuperar la propiedad de montículo, se pone en su lugar el último elemento del montículo (el de más profundidad y más a la derecha en el árbol binario) y se realiza sobre el mismo la función *Hundir*.

```
fun ObtenerCima(m: montículo): elemento
    var
        e:elemento
    fvar
        si m.c ≠ 0 entonces
            e ← m.T[1]
            m.T[1] ← m.T[m.c]
            m.c ← m.c - 1
            Hundir(m.T,1)
            dev e
        fsi
    ffun
```

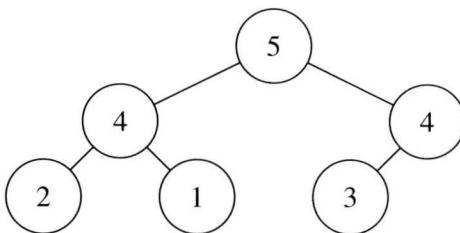
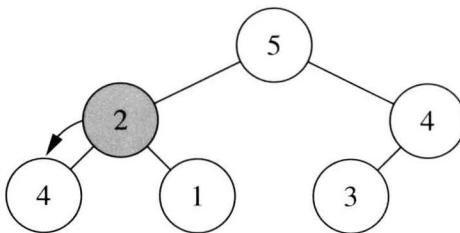
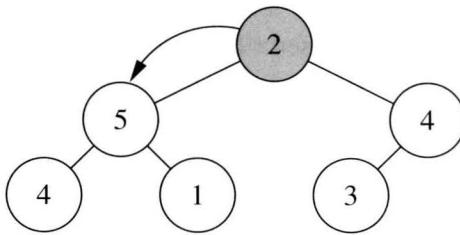
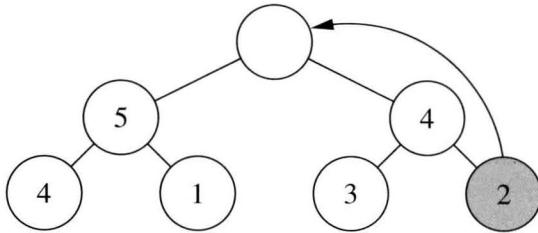
MONTÍCULOS

Figura 2.22: Operación de la recuperación de la propiedad de montículo tras el borrado de la cima

2.2.2 Eficiencia en la creación de montículos a partir de un vector

La creación de un montículo a partir de una colección de valores puede optimizarse para tener un coste lineal. Veamos algunos procedimientos para realizar esta operación y analicemos su eficiencia.

Función *CreaMonticulo*

La primera de las opciones de creación del montículo a partir de un vector se puede realizar utilizando la función *flotar*, que inserta cada elemento al final del montículo y recupera la propiedad del mismo [BB90].

Para crear un montículo a partir de un vector $V[1..n]$ mediante inserciones, usamos una simple iteración:

```
fun CreaMonticulo(T:vector[1..n]): monticulo
  var
    m: monticulo
  fvar
    m  $\leftarrow$  CreaMonticuloVacio();
    para i  $\leftarrow$  2 hasta n hacer
      Flotar(T,i);
    fpara
    dev(m)
ffun
```

Si n el número de elementos del montículo, la profundidad media es $\log_2(n) - 1$, lo que equivale a una complejidad $O(n \log n)$. El i -ésimo objeto requerirá como mucho $O(\lceil \log(i) \rceil)$ intercambios en el árbol, por lo que el tiempo de ejecución $T(n)$ será:

$$T(n) \leq \sum_{i=1}^n c \lceil \log(i) \rceil = O(n \log n)$$

Sin embargo hay una manera más eficiente de crear un montículo a partir de los elementos de un vector mediante el procedimiento *Hundir*:

```
fun CreaMonticuloLineal(T:vector[1..n]): monticulo
  var
    m:monticulo
  fvar
    m  $\leftarrow$  CreaMonticuloVacio();
    para i  $\leftarrow$  n/2 hasta 1 paso -1 hacer
      hundir(T,i);
    fpara
    m.T  $\leftarrow$  T
ffun
```

dev(m)
ffun

El último nivel del árbol contiene los $n/2$ últimos elementos del vector, y éstos no pueden hundirse más, por lo que se parte del penúltimo nivel, es decir desde $i = n/2$. Vamos a calcular el coste de este algoritmo suponiendo sin pérdida de generalidad que n es potencia de 2, es decir que el árbol binario es completo. De esta manera, si $n = 2^k$ habrá $n/2 = 2^{k-1}$ hojas que ya son un montículo. A partir de ese nivel habrá $n/4 = 2^{k-2}$ nodos que requerirán una iteración del bucle y una operación *hundir*, $n/8 = 2^{k-3}$ nodos requerirán como mucho dos iteraciones, y así sucesivamente. En general $n/2^{i+1} = 2^{k-i-1}$ nodos requerirán como mucho i iteraciones en el bucle. Así el número de iteraciones totales por el bucle es:

$$\sum_{i=1}^{k-1} i2^{k-i-1}$$

Podemos expandir la expresión:

$$\sum_{i=1}^{k-1} i2^{k-i-1} = 1 \cdot 2^{k-2} + 2 \cdot 2^{k-3} + \dots + (k-2)2^1 + (k-1)2^0$$

de esta manera vemos que aparece una vez el término 2^{k-2} , dos veces el término 2^{k-3} , etc. Reordenando la expresión tenemos:

$$\begin{aligned} \sum_{i=1}^{k-1} i2^{k-i-1} &= (2^{k-2} + 2^{k-3} + 2^{k-4} + \dots + 2^1 + 2^0) \\ &\quad + (2^{k-3} + 2^{k-4} + \dots + 2^1 + 2^0) \\ &\quad + (2^{k-4} + \dots + 2^1 + 2^0) + \dots + 2^0 \end{aligned}$$

Sabiendo que:

$$\sum_{i=1}^k 2^{i-1} = 2^{k-1} - 1$$

las sub-expresiones anteriores suman respectivamente $2^{k-1} - 1$, $2^{k-2} - 1$, etc. Y por tanto, considerando, como hemos dicho antes, que $n = 2^k$ nos queda:

$$\begin{aligned} \sum_{i=1}^k i2^{i-1} &= (2^{k-1} - 1) + (2^{k-2} - 1) + \dots + (2^1 - 1) \\ &= (2^{k-1} + 2^{k-2} + 2^{k-3} + \dots + 2) - (k-1) \end{aligned}$$

$$= (2^{k-1} + 2^{k-2} + 2^{k-3} + \dots + 2 + 1 - 1) - (k - 1)$$

$$= 2^k - 1 - (k - 1) = 2^k - k - 1 = n - \log n - 1$$

y la expresión $n - \log n - 1 \in O(n)$

Ordenación basada en montículos: Algoritmo *Heapsort*

El concepto de montículo como estructura de datos fue desarrollado *ad hoc* como apoyo a la creación de un algoritmo de ordenación eficiente. Con un montículo disponemos de una estructura de datos en la que encontrar el mínimo (o el máximo) es una operación de coste constante. Una vez extraído el primer elemento, restaurar la propiedad de montículo tiene un coste $O(\log n)$. Sin pérdida de generalidad, asumimos que vamos a usar montículos de máximos. El funcionamiento del algoritmo es como sigue:

- Se convierte el vector en un montículo.
- Se selecciona el máximo (la cima del montículo) y se incorpora al vector solución S . El montículo pierde un elemento y el vector S lo incorpora.
- Se restaura la propiedad del montículo sobre los elementos que restan. Así estamos en condiciones de volver a seleccionar el máximo.
- Al finalizar el vector S contiene el vector de entrada ordenado de mayor a menor.

Es decir:

```
fun Heapsort(T:vector[1..n] de entero): vector[1..n] de entero
  var
    e:entero
    M:monticulo
    S: vector[1..n]
  fvar
    M  $\leftarrow$  CreaMonticuloLineal(T);
    para i  $\leftarrow$  1 hasta n hacer
      e  $\leftarrow$  ObtenerCima(M)
      S[i]  $\leftarrow$  e
    fpara
    dev S
ffun
```

El coste se basa en la creación y restauración de la propiedad de montículo en cada extracción de la cima (primer elemento) del montículo. La creación es de coste lineal

como hemos visto antes. En cada paso del bucle se extrae el primer elemento y se restaura la propiedad de montículo con un coste total $O(\log n)$. Como realizamos n operaciones de coste $O(\log n)$, el algoritmo tiene un coste global $O(n \log n)$.

2.2.3 Otros tipos de montículos

El montículo que describimos en este capítulo se denomina montículo binario y como hemos indicado se usa para la implementación de colas de prioridad y que son de utilidad en otros algoritmos, además hay otros tipos de montículos, entre los que destacan los siguientes:

Montículo binomial

Un montículo binomial es una colección de árboles binomiales que se definen de manera recursiva:

- En el caso de un único elemento, un árbol binomial M_0 de orden 0 es un nodo.
- Un árbol binomial M_k de orden k tiene una raíz de grado k y los hijos son raíces de árboles binomiales de orden $k - 1, k - 2, \dots, 0$.

De esta manera, es posible construir un árbol binomial de orden k a partir de dos árboles de orden $k - 1$ agregando uno de ellos como el hijo más a la izquierda del otro. En general, un árbol binomial de orden k contendrá $2k$ nodos, y tendrá una altura k .

Los árboles de un montículo binomial tienen las siguiente propiedades:

- Los hijos son siempre menores (mayores) o iguales que el padre para un montículo de mínimos (máximos).
- Para todo $k > 0$ existe al menos un árbol binomial en el montículo con una raíz de grado k .

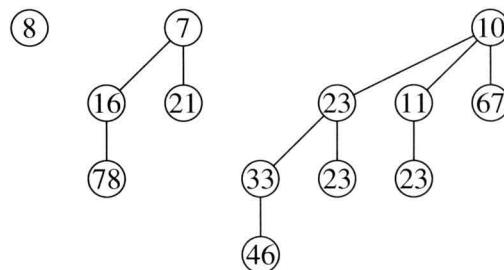


Figura 2.23: Un montículo binomial compuesto por los subárboles M_0 , M_1 y M_2

Cada subárbol del montículo binomial cumple que:

- La raíz del subárbol tiene en la cima el elemento mínimo (máximo).
- Puede haber solo 0 ó 1 árbol binomial por cada orden, incluido el orden 0.

Como se ve en la figura 2.23, el montículo binomial se implementa mediante una serie de árboles binomiales que cumplen por separado la propiedad de montículo. Un montículo binomial con n elementos consiste en al menos $\log n + 1$ árboles binomiales, cada uno de ellos corresponde al dígito de la representación binaria de n . Por ejemplo, un montículo binomial de 13 elementos se compone de 3 árboles binomiales de ordenes 3, 2 y 0, al ser $13_{10} = 1101_2$. Los montículos binomiales mejoran la eficiencia frente a una implementación con montículos básicos.

Montículo de fibonacci

Un montículo de *fibonacci* es una colección de árboles que al igual que los anteriores, satisfacen cada uno de ellos la propiedad de montículo. La estructura es más flexible que en el caso de los montículos binomiales, ya que no restringe la existencia o no de árboles o la profundidad de los niveles (puede haber un montículo de *fibonacci* con todos sus elementos en un solo árbol). Esta flexibilidad va a permitir realizar operaciones como la mezcla de montículos, que se realiza concatenando las dos listas de árboles de cada montículo.

Las condiciones para ser montículo de *fibonacci* son que el número de hijos de cada nodo (grado del nodo) se mantiene menor de $\log n$, y el tamaño de los subárboles con raíz en un nodo de grado k es al menos F_{k+2} , siendo F_k el k -ésimo número de la serie de *Fibonacci*.

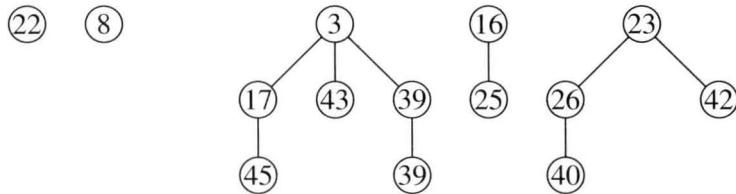


Figura 2.24: *Un montículo de fibonacci*

La raíz de un montículo de *fibonacci* es el menor elemento de primer nivel (el valor 3 en la figura 2.24). Para la implementación de operaciones se utilizan listas circulares por niveles. El primer nivel está formado por las raíces de los árboles (elementos 22, 8, 3, 16, 23) que forman la primera lista circular. El resto de las listas son las listas formadas por los hijos del mismo nivel de cada uno de los árboles (los elementos 17, 43, 39 del subárbol de raíz 3, elemento 25 de raíz 16, elementos 26, 42 del subárbol de raíz 23 etc.

A modo de referencia se detallan algunos costes de operaciones conocidas para ilustrar la eficiencia de este tipo de estructuras.

| | Lista enlazada | Montículo | Montículo binomial | Montículo de fibonacci |
|-----------------|----------------|-------------|--------------------|------------------------|
| insertar | $O(1)$ | $O(\log n)$ | $O(\log n)$ | $O(1)$ |
| mínimo (máximo) | $O(n)$ | $O(1)$ | $O(\log n)$ | $O(1)$ |
| borrar | $O(n)$ | $O(\log n)$ | $O(\log n)$ | $O(\log n)$ |

El montículo de *fibonacci* mejora la complejidad de algoritmos como el de Dijkstra y el de hallar el árbol de recubrimiento mínimo.

2.3 Tablas de dispersión (*Hash*)

Las tablas *Hash* o tablas de dispersion permiten el almacenamiento de datos pertenecientes a un dominio potencialmente muy grande de registros, pero del que sólo usamos y referenciamos un número reducido (figura 2.25). El uso de tablas *Hash* permite un ahorro considerable de memoria y es eficiente, siempre que el número de registros del subconjunto se mantenga dentro de lo previsto. Es decir, en lugar de organizar un fichero mediante el almacenamiento por clave directa del dominio inicial, reducimos el espacio de claves y ubicamos los registros necesarios para almacenar el subconjunto. Para lograr la reducción del espacio de claves se hace corresponder a cada clave directa, una clave en el subconjunto de índices de la tabla *Hash*, mediante una *función Hash*. Considerando lo anterior, entra dentro de lo previsible que dos o más claves diferentes del dominio inicial, proporcionen la misma clave en el índice de la tabla *Hash*, lo que se conoce como *colisión*.

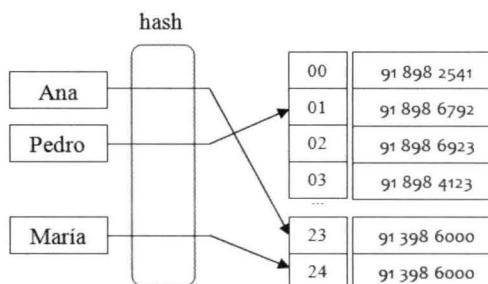


Figura 2.25: Una función *Hash* para una agenda

Un ejemplo conocido de uso es el de los procesos de búsqueda indexada lineal, donde se realiza una búsqueda ordinaria consistente en un recorrido y comparación con las claves del arreglo hasta encontrar el elemento buscado. En este tipo de búsqueda usamos generalmente como claves los índices del array o de los registros del fichero. Sin embargo la búsqueda con estas condiciones no es frecuente, y tampoco lo es disponer de un rango de claves del mismo tamaño que el del espacio de búsqueda.

Al contrario que en el procedimiento de búsqueda basado en arrays e índices naturales, ahora una clave puede dar lugar a un rango de elementos asociados, y no a uno solo.

Más formalmente podemos definir una *tabla Hash* como un conjunto formado por un número de elementos variable y del mismo tipo, que se asocian a una única palabra clave para un grupo de posibles valores, más una serie de procedimientos de acceso basados en *funciones Hash*. La función *Hash* asocia claves a valores *Hash*, que a su vez son usados como índices en la tabla *Hash*.

Se aprecian por tanto dos diferencias fundamentales frente a la búsqueda por clave directa que son:

- El índice o clave permite encontrar o asociar (*mapping*) un *rango* de elementos.
- La búsqueda no *compara* por valores clave directamente, sino que aplica una función $h(k)$ que nos da directamente la localización de la clave k en la estructura o tabla *Hash*.

Ahora, en vez de navegar por la estructura comparando palabras clave con las claves en los elementos, accedemos a los elementos de la tabla haciendo operaciones algorítmicas que transforman las claves en direcciones de la tabla. Estas operaciones algorítmicas son llevadas a cabo por la función *Hash*. Las funciones *Hash* son por tanto funciones que transforman claves (llamadas *llaves*) en direcciones que referencian (aunque no de manera unívoca) a un documento o registro en la tabla o estructura donde están almacenados. Se da por hecho que la estructura no es referenciable directamente, es decir, que el espacio de claves es mucho mayor que el de los registros posibles, y que a su vez éstos no ocurren o existen en todos los casos.

Un ejemplo de tabla *Hash* es un fichero de datos de personas indexados por DNI. Si tenemos una tabla *Hash* de 10^3 bloques para el almacenamiento y búsqueda de estos datos, el conjunto de claves posibles es el dominio de DNIs que va del 00000000 hasta el 99999999, es decir $[0 \dots 10^8 - 1]$ y el conjunto de índices de la tabla *Hash* es $[0 \dots 10^3 - 1]$.

2.3.1 Funciones Hash

Una función *Hash* asocia una clave con una posición en la tabla *Hash*. La función *Hash* es una aplicación no necesariamente inyectiva entre el conjunto dominio de las claves X y el conjunto dominio de direcciones D de la estructura de datos:

$$H : X \longrightarrow D$$

$$x \longrightarrow h(x)$$

Las funciones $h(x)$ deben tener las siguientes propiedades:

- Deben repartir equiprobablemente los valores, es decir, para una distribución de probabilidad determinada de valores de X , los valores $h(x)$ en D deben mantener dicha distribución. En este sentido, la distribución de salida no debe estar ligada a ningún patrón.
- La función debe poderse calcular de manera eficiente, de forma que en ningún caso sea el cálculo de $h(x)$ lo que determine la complejidad del proceso de búsqueda.
- Cambios en la clave, aun siendo pequeños, deben resultar en cambios significativos en la función *Hash*.

Conceptualmente una función *Hash* es un pseudo-generador de números aleatorios sobre un rango determinado. Sus propiedades se analizan por tanto también desde el punto de vista estadístico, estudiando el tipo de distribución que realizan para determinados valores de entrada, y cómo se comporta la salida ante variaciones de la entrada; por ejemplo para variaciones pequeñas de la entrada, una función Hash no debe en general reproducir el patrón en la salida.

Hay básicamente dos dimensiones sobre las que podemos evaluar funciones *Hash*:

Distribución de datos: estudiamos de qué manera y mediante qué distribución estadística se distribuyen los valores dentro del conjunto de datos. Esto implica analizar también las distribuciones de probabilidad de las posibles colisiones.

Eficiencia: mide la eficiencia de cálculo de la función Hash. Una función Hash debe ser estable, rápida y determinista y en general es aceptado que el valor medio de complejidad debe ser constante, o a lo sumo de orden logarítmico.

Una de las primeras cuestiones que se plantean es la de cómo construir las funciones apropiadas a cada tabla Hash. En general, construir funciones $h(x)$ que cumplan con las propiedades que buscamos no es trivial. La dificultad no es tanto la de evitar las colisiones, como de repartir éstas adecuadamente. En cuanto a la frecuencia de las colisiones podemos mencionar un ejemplo conocido como el de la paradoja del cumpleaños. Dada una muestra de N personas, la probabilidad p de que entre ellas haya dos con la misma fecha de nacimiento crece muy rápidamente para un N aparentemente pequeño hasta pasar a ser $p > 0.5$ con tan solo $N > 23$. Esto significa que aplicando 23 claves a una tabla con 365 valores, las probabilidades de colisión en caso de que la función Hash sea equiprobable son de $1/2$.

En consecuencia, las aplicaciones $h(k)$, tienen la particularidad de que podemos esperar que $h(k_i) = h(k_j)$ para bastantes pares distintos, con lo que el otro aspecto del problema será el de diseñar métodos de resolución de colisiones cuando éstas se produzcan.

Propiedades de las funciones *Hash*

Las funciones *Hash* son útiles en la medida que proporcionan un mecanismo de direccionamiento que cumple una serie de requisitos. Básicamente nos centraremos en estas propiedades:

Sentido único: las funciones *Hash* son funciones que en general no mantienen en el resultado la semejanza entre dos entradas. Esto es especialmente útil en el uso de funciones *Hash* para el cálculo de huellas criptográficas. La utilidad de la huella digital de un documento es la de proporcionar un resumen de un tamaño fijo, de manera que su tratamiento sea homogéneo sea cual sea el tamaño del documento original. Sin entrar en detalles, esto permite por ejemplo firmar electrónicamente solo la huella, y que baste con ello para asegurar la autoría del documento. Pero solo garantizamos que es así cuando desde el punto de vista matemático la probabilidad de que dos entradas diferentes den como resultado la misma clave *Hash* es nula. Por otro lado, la función *Hash* en aplicaciones criptográficas debe garantizar que no es posible de ninguna manera reconstruir el documento original a partir de la huella. No ocurre así en otras aplicaciones de este tipo de funciones.

Colisiones: cualquier función *Hash* debe evitar las colisiones entre valores, pero por definición las colisiones son inevitables ya que el espacio de valores de salida es mucho menor que el espacio de valores de la entrada. La manera de minimizar la ocurrencia de colisiones es conseguir que la distribución de valores de salida sea uniforme con respecto a la distribución de entrada.

Distribución de datos: esta medida determina la bondad de la función Hash en distribuir uniformemente los valores de los elementos en el conjunto de los índices de la tabla. El análisis de esta medida requiere conocer previamente el número de colisiones que ocurren con el conjunto de datos de entrada.

Tipos de funciones Hash

No hay un único criterio a la hora de definir funciones *Hash* y éste varía según el dominio de las claves (natural, entero, string, etc.) y la uniformidad del dominio destino. Es necesario por tanto un análisis del espacio de claves para asegurar la elección o creación de una función Hash adecuada a los propósitos que se buscan.

Se enumeran a continuación algunas de las funciones Hash más usuales.

Función Módulo: la función módulo calcula el resto de la división de la clave por un valor $M + 1$, siendo M el número de índices existentes en la tabla *Hash*.

$$h(k) \longrightarrow (k \bmod (M+1))$$

La elección de M es bastante importante, ya que un valor poco apropiado puede hacer no solo que se elimine la uniformidad, sino que haya valores que no se obtienen nunca. En el caso de que el dominio de las claves no esté distribuido uniformemente, se pueden producir situaciones no deseables en el resultado. Por ejemplo en el caso de que en el dominio de las claves sean frecuentes valores terminados en 0, una elección de M del tipo $M = 10^k$ es poco apropiada. En general, si no conocemos la distribución del conjunto de claves, conviene una elección de M primo.

Podemos ilustrarlo con un ejemplo. Sea $N = 63$ el tamaño de la tabla definida entre 1 y 63. Sean $k_1 = 7259$ y $k_2 = 8708$ dos claves que deban almacenarse en el array. Si se aplica la formula con $N = 63$ queda:

$$h(k_1) = 7259 \bmod 63 = 14$$

$$h(k_2) = 8708 \bmod 63 = 14$$

En este caso, como $h(k_1) = h(k_2)$ (y $k_1 \neq k_2$) se está ante una colisión. Si se aplica la formula con $N=64$ no ocurre la colisión, ya que se tiene:

$$h(k_1) = 7259 \bmod 64 = 27$$

$$h(k_2) = 8708 \bmod 64 = 4$$

En este método hay que tener en cuenta el dominio de claves que manejamos. Si éste es no uniforme (por ejemplo palabras de un idioma), estamos expuestos a consecuencias no deseadas, como la coincidencia de valores de la función *Hash* para las palabras más probables. Por ejemplo, en el caso de caracteres de 8 bits si $M = 2^k$ para $k = 24$, las cadenas que tengan los 24 últimos bits iguales, es decir que comparten los 3 últimos caracteres, se asignan a la misma dirección en la tabla *Hash*, siendo alto el número de colisiones.

Función Cuadrado: este método eleva al cuadrado el valor de la clave y elimina los c bits centrales del resultado, dando un valor en el rango $[0..2^c - 1]$. Por ejemplo, consideremos registros cuya clave sean valores de 4 dígitos numéricos en base 10, y hacemos corresponder este espacio de claves con una tabla Hash de tamaño 100 de rango $[0..99]$.

En este ejemplo, sea $c = 2$. Si la clave es 5678 el cuadrado es el número de 8 dígitos 32239684 y sus dos dígitos centrales son 39. Todos los dígitos de la clave original han contribuido a obtener el valor de sus dígitos centrales, lo que garantiza que el resultado no está influenciado por una parte de la clave original (por ejemplo de sus dígitos más a la izquierda o más a la derecha).

Función plegado (compresión): Este método toma partes de la clave y opera sobre ellas, por ejemplo realizando operaciones *XOR* por cada uno de los dígitos. El proceso puede ser muy rápido y aplicarse no solamente a valores numéricos, sino también alfanuméricos, aunque puede tener consecuencias no deseadas, como por ejemplo que una permutación de los elementos dé lugar a colisiones.

Un ejemplo sencillo puede ser la operación sobre claves alfanuméricas, realizando la operación *XOR* entre los caracteres

$$\begin{aligned}
 h(UNED) &= U \oplus N \oplus E \oplus D \\
 &= (01010101 \oplus 01001110) \oplus (01000101 \oplus 01000100) \\
 &= 00011111 \oplus 00000001 \\
 &= 00011110
 \end{aligned}$$

Función multiplicación: el método opera en dos pasos. En primer lugar se normaliza el valor de entrada multiplicándolo por una cantidad $\phi \in (0..1)$ y seguidamente multiplicamos éste valor por el numero M de entradas de la tabla, truncando la parte fraccionaria (representado por **mod1**).

$$h(k) = \lfloor M(k\phi \textbf{mod1}) \rfloor$$

Algunos valores de ϕ funcionan mejor que otros. En concreto es apropiada la cantidad $\phi_a = \frac{\sqrt{5}-1}{2}$, denominada *razón áurea*.

Por ejemplo supongamos $\phi = 0.618$ y $M = 1000$, obteniendo

$$h(3) = \lfloor 1000((0.618 \times 3)\textbf{mod1}) \rfloor = \lfloor 1000((1.854)\textbf{mod1}) \rfloor = 854$$

Otros valores serían $h(1) = 618$, $h(4) = 472$, $h(5) = 90$, etc.

2.3.2 Resolución de colisiones

Al contrario que en el caso de la criptografía de clave publica, donde una función *Hash* debe cumplir la propiedad de que dados dos documentos distintos d y d' , la probabilidad de que $h(d) = h(d')$ sea prácticamente nula, en el caso de funciones *Hash* para la ordenación, el caso de que dos claves *colisionen* es más frecuente, aunque se trata de evitar igualmente y se procura que la colisión sea equiprobable. Igualmente, hay que tener en cuenta qué hacer en caso de conflicto, lo que se conoce como *resolución de colisiones*.

En este caso se requiere almacenar los registros que comparten clave de manera que en caso de colisión, sean accesibles igualmente.

Hay dos métodos principales: *hashing abierto* y *hashing cerrado*. La elección de uno u otro depende de factores como el *factor de carga*, que es un parámetro que determina cuántas posiciones de la tabla se han ocupado y la probabilidad de encontrar una entrada vacía. El factor de carga se define como:

$$\delta = \frac{n}{M}$$

siendo M el tamaño de la tabla, y n el número de índices ocupados. El factor de carga es por tanto un valor entre 0 (vacía) y 1 (llena) que mide la proporción de la tabla Hash ya ocupada. Normalmente es necesario extender la tabla con entradas cuando se ha superado el factor 0.5. Si no es así, lo que se realiza en caso de colisión es un recorrido de entradas vacías hasta encontrar una.

Hashing abierto

Utiliza estructuras dinámicas externas a la tabla para el almacenamiento de las claves que han generado colisiones. Desde el punto de vista conceptual el método es válido, pero si consideramos la eficiencia, el recorrido lineal en éstas estructuras no es en ocasiones tan eficiente como se podría esperar de seguir en la misma estructura de datos.

Hashing cerrado

El *hashing cerrado* permite resolver la colisión mediante la búsqueda en ubicaciones alternativas en la misma tabla, hasta que encontramos un sitio libre en la misma. Se debe determinar que hay un sitio libre en la tabla con la presencia de un valor que lo determine, y si es así, se ubica el valor en la posición indicada por la función Hash. En este tipo de direccionamiento, a medida que la tabla se llena, las probabilidades de colisión aumentan significativamente.

Los métodos más conocidos para implementar este tipo de resolución de colisiones son:

Recorrido lineal: en general, a la dirección obtenida por la función Hash $h(k)$ se le añade un incremento lineal que proporciona otra dirección en la tabla:

$$s(k, i) = (h(k) + i) \bmod m$$

siendo m el tamaño de la tabla.

Recorrido cuadrático: en el caso del recorrido lineal, la probabilidad de nuevas colisiones es bastante alta para determinados patrones de claves. Hay otro método basado en una expresión cuadrática $g(k)$ que permite mayor dispersión de las colisiones por la tabla, al mismo tiempo que proporciona un recorrido completo por la misma. Así, si usamos:

$$g(i) = i^2$$

la i -ésima colisión para un valor k viene dada por la secuencia $h(k), h(k) + c_1 i^2, h(k) + c_2 i^2, \dots$ es decir:

$$dir = (h(k) + c_k g(k)) \bmod m$$

Las constantes pueden ser del tipo $c_k = 1$ o diferentes. En el caso de que $m = 4j + 3$ con j entero, entonces se cumple que la exploración cuadrática es completa y ésta recorre todos los bloques de la tabla.

Recorrido mediante doble Hashing: en el caso del doble Hashing, se utiliza una segunda función *Hash* $h'(x)$ como función auxiliar. De esta manera tenemos:

$$dir = h(k) + ch'(k)$$

con c constante.

Para que este método funcione, la función $h'(x)$ debe cumplir determinadas condiciones:

- $h'(x) \neq 0$, ya que de lo contrario se producen colisiones.
- La función h' debe ser diferente de la función h .
- Los valores de $h'(x)$ deben ser primos relativos de M (es decir, que no comparten factores primos), para que los índices de la tabla se ocupen en su totalidad. Si M es primo, cualquier función puede ser usada como h' .

Como ejemplo de doble Hashing y siendo M primo, podemos considerar en general la función $h(x) = K - (x \bmod K)$ siendo K un número primo menor que M .

El número medio de comprobaciones de búsqueda depende del método de recorrido utilizado y del factor de carga. La tabla siguiente define el coste de las comprobaciones:

| TIPO | EXITOSA | NO EXITOSA |
|--------------------|---|---|
| Exploración lineal | $\frac{1}{2}(1 + \frac{1}{(1-\delta)^2})$ | $\frac{1}{2}(1 + \frac{1}{(1-\delta)^2})$ |
| Doble Hashing | $-\frac{\ln(1-\delta)}{\delta}$ | $\frac{1}{1-\delta}$ |
| Sondeo cuadrático | $-\frac{\ln(1-\delta)}{\delta}$ | $\frac{1}{1-\delta}$ |

El desarrollo de funciones *Hash* no siempre es fácil. En primer lugar por la dificultad de crear una estructura de datos que se adapte a las necesidades, ya que los comportamientos están diferenciados según la aplicación que tengan, como por ejemplo en el caso de

aplicaciones criptográficas o en el uso en estructuras de almacenamiento. En segundo lugar por garantizar la eficiencia y la necesidad de asegurar la distribución uniforme de los valores de salida de acuerdo con propiedades estadísticas de los valores de entrada. En algunos casos, este coste de eficiencia puede dar lugar a que se usen estructuras de tipo árbol binario en lugar de tablas Hash.

2.4 Ejercicios propuestos

1. Dado un grafo comprueba si tiene ciclos o no. También se denomina prueba de aciclicidad.
2. Escribe un algoritmo que realice una ordenación topológica de un gda, pero de manera que sólo se visite un vértice cuando se han visitado todos sus predecesores.
3. Demuestra que si AR es un árbol de recubrimiento para un grafo no dirigido G , entonces al añadir una arista a a AR tal que antes no formaba parte del conjunto de aristas de AR y sí formaba parte del conjunto de aristas de G , crea un único ciclo.
4. Demuestra que todo grafo no dirigido conexo contiene un nodo que al eliminarlo el grafo sigue siendo conexo. Diseña un algoritmo que encuentre dicho nodo con un coste de orden similar al de los recorridos de un grafo.
5. ¿Cuál es la conectividad máxima de un grafo de n nodos?
6. ¿Cuántas aristas debe tener un grafo como mínimo para ser biconexo?
7. ¿Cuántas aristas debe tener un grafo como mínimo para ser biconexo?
8. Implementar la función *Flotar* de manera recursiva.
9. Implementar la función *Hundir* de manera recursiva.
10. Considere un montículo de enteros implementado como un array T y un contador c . Escribir un método que compruebe que el montículo está organizado correctamente y halla el coste.
11. Dado el vector $m = [10, 12, 1, 14, 6, 5, 8, 15, 34, 1, 9, 4, 23, 14, 51, 6, 8, 3, 5]$ realizar la traza completa, señalando los intercambios en el vector y el nivel al que se producen. Evaluar el coste de realizar la creación del montículo con las funciones *CreaMonticulo()* y *CreaMonticuloLineal()*.
12. Dado el siguiente algoritmo:

```
fun montificar(T:vector[1..N],i)
    si (2*i + 1 ≤ N/2) entonces montificar(2*i+1) fsi
```

```
si ( $2 * i \leq N/2$ ) entonces montificar( $2*i$ ) fsi  
hundir(i)  
ffun
```

Explicar el resultado del algoritmo para la llamada *montificar*($T, N/2$) y calcular el coste del mismo.

13. Explica cómo se halla el elemento mínimo en un montículo de máximos. Calcula el coste de dicha operación.
14. En un montículo de máximos no es necesario recorrer todo el montículo para calcular el elemento máximo. ¿Es cierta la afirmación? En caso afirmativo indique la complejidad de la operación.
15. A la hora de implementar el método de construcción de un montículo se obtiene una complejidad $O(n \log 2n)$. ¿Es mejorable? ¿Como?
16. Dada la entrada 4371, 1323, 6173, 4199, 4344, 9679, 1989 y una función Hash(x)= $x \bmod 10$, dar la correspondiente:
 - (a) tabla *Hash* por recorrido lineal.
 - (b) tabla *Hash* por recorrido cuadrático.
 - (c) tabla *Hash* con doble *hashing* usando la función $h2(x) = 7 - (x \bmod 7)$.
17. Tenemos un diccionario de 30.000 palabras que queremos guardar en una tabla *Hash* con *hashing* cerrado. Suponiendo que el tamaño medio de una palabra es siete letras y que podemos guardar palabras de m letras en $m + 1$ bytes, cuánta memoria necesitamos?
18. Empleamos una tabla *Hash* con exploración cuadrática para almacenar 10.000 objetos de tipo cadena. Supongamos que el factor de carga es $\delta = 0.4$, que la longitud mínima de las cadenas es 8, que cada carácter ocupa 2 bytes y que una referencia ocupa 4 bytes. Determinar:
 - El tamaño de la tabla *Hash*.
 - La cantidad de memoria empleada para almacenar los 10.000 objetos de tipo cadena.
 - La cantidad de memoria adicional empleada por la tabla *Hash*.
 - La memoria total que necesita la tabla *Hash*.
19. Se tiene una tabla de tamaño 13 donde se desean insertar los elementos 54, 99, 83, 33, 139, 126, 143, 91, 173 y 41. Se utilizará un doble *hashing* con funciones $h(x) = x \bmod 13$ y $h'(x) = 1 + (x \bmod 11)$. Se pide describir los pasos para la inserción de estos elementos.

20. Se tiene una tabla de tamaño $M = 2^k$ y por tanto M no es primo. ¿Cómo definir una función *Hash* secundaria válida?
21. Sea T una tabla de *Hash* de tamaño 10 y sea $h(k) = 4 + 3k \bmod 10$. Se quieren insertar en T los elementos con claves 1, 11, 5, 15, 55, 6, 26, 90, 50, 20 en ese mismo orden usando h . Se pide: (a) Determinar el resultado de insertar las claves en T si las colisiones se resuelven por recorrido lineal. (b) Determinar el resultado de insertar las claves en T si las colisiones se resuelven por recorrido cuadrático.

2.5 Notas bibliográficas

Dos referencias clásicas para estudiar las estructuras de datos y en particular los grafos son [AHU98] y [HS94]. Aunque en este último los algoritmos están escritos en Pascal, dadas las características de este lenguaje se entienden sin ningún problema. En [BB06] también se estudian los grafos, al igual que en [GMCLMPGC03] y [MOOMV03]. En [MOOMV03] se puede encontrar la especificación del TAD grafo, la implementación de las operaciones básicas y varios ejercicios resueltos. En todas las demás referencias se encuentran numerosos ejercicios propuestos. Un estudio más profundo de la teoría de grafos se puede encontrar en [ST81] y [Chr75]. En las referencias anteriores y en [GMCLMPGC03] se pueden encontrar las soluciones de la mayoría de los ejercicios propuestos sobre grafos.

El montículo fue desarrollado en [Wil64] y [Flo67] para la implementación de la primera versión de su algoritmo de ordenación *Heapsort*. La estructura de datos como tal, fue de hecho diseñada *ad hoc* para dicho algoritmo y usada posteriormente para el desarrollo de colas de prioridad en otros algoritmos conocidos. Los montículos de *fibonacci* aparecen por primera vez en [FT87]. El desarrollo de las tablas *Hash* está descrito en [Knu73].

Capítulo 3

Algoritmos voraces

Este capítulo abre el estudio de los esquemas algorítmicos. Los esquemas algorítmicos son familias generales de algoritmos que comparten el mismo enfoque de resolución, y que permiten resolver problemas concretos mediante la instanciación de los elementos del esquema al problema.

El principal objetivo de este capítulo es presentar el esquema voraz y los tipos de problemas más representativos que es capaz de resolver. Tras la lectura de este capítulo, el lector debería ser capaz de identificar si un problema concreto se puede resolver utilizando este esquema, de instanciar los elementos del esquema a las características específicas del problema, razonar sobre el coste computacional y demostrar la optimalidad de la solución propuesta.

Se recomienda al lector una lectura secuencial del capítulo. Cuando vaya a abordar los problemas resueltos, que corresponden a los problemas más representativos de este esquema, se propone al lector que intente identificar los elementos característicos del esquema antes de consultar la solución. También se recomienda realizar la traza del algoritmo propuesto, así como una lectura detenida de las demostraciones de optimalidad. Finalmente, el lector puede intentar resolver los problemas propuestos cuya solución se encuentra en diferentes fuentes bibliográficas a las que se hace referencia al final del capítulo.

3.1 Planteamiento general

El esquema voraz (*greedy algorithms*) se aplica a problemas de optimización en los que la solución se puede construir paso a paso sin necesidad de reconsiderar decisiones ya tomadas. Genéricamente el problema que se puede resolver con este tipo de esquema es: *encontrar un conjunto de candidatos que constituya una solución y que optimice una función objetivo*. Este esquema se utiliza principalmente en problemas de planificación de tareas y en problemas que se pueden modelar con grafos, en los que hay que realizar una búsqueda, cálculo de recorridos u optimización de pesos, entre otras tareas.

Los problemas que se pueden resolver con este esquema constan de n candidatos y se trata de encontrar una solución basada en hallar un subconjunto de esos candidatos, o una secuencia ordenada de los mismos, de manera que se optimice (maximice o minimice) una función objetivo. En este esquema se trabaja por etapas, considerando la elección de un candidato en cada etapa. Habrá que seleccionar en cada una el candidato más prometedor de los aún disponibles y decidir si se incluye o no en la solución.

Las características y elementos que pueden intervenir en este esquema son los siguientes:

- Resolución de un problema de forma óptima.
- *Conjunto inicial de candidatos* (elementos que hay que planificar, vértices o aristas de un grafo, ...).
- *Conjunto de candidatos* que ya han sido considerados y *seleccionados*. Inicialmente este conjunto está vacío.
- *Conjunto de candidatos* que ya han sido considerados y han sido *rechazados*, de manera que ya no volverán a ser considerados.
- *Función* que determina si un conjunto de candidatos es una *solución* al problema.
- *Función factible*, que determina si un conjunto es completable o factible; es decir, si añadiendo nuevos candidatos se puede alcanzar una solución que cumple las restricciones del problema.
- *Función de selección* que escoge al candidato más prometedor de los que todavía no se han considerado.
- *Función objetivo*, representa el coste o valor de una solución (tiempo de proceso, longitud del camino, ...) y es la que se quiere optimizar (maximizar o minimizar). Esta función no tiene porqué aparecer explícitamente en el algoritmo.

El esquema genérico de los algoritmos voraces es el siguiente:

```
fun Voraz(c: conjuntoCandidatos): conjuntoCandidatos
    sol ←  $\emptyset$ 
    mientras  $c \neq \emptyset \wedge \neg \text{solucion}(sol)$  hacer
         $x \leftarrow \text{seleccionar}(c)$ 
         $c \leftarrow c \setminus \{x\}$ 
        si factible( $sol \cup \{x\}$ ) entonces
             $sol \leftarrow sol \cup \{x\}$ 
        fsi
    fmientras
    si solucion( $sol$ ) entonces devolver  $sol$ 
    sino imprimir('no hay solución')
```

fsi
ffun

En el esquema general aparecen los conjuntos y funciones antes mencionados:

- c es el *conjunto de candidatos*.
- sol es el *conjunto de candidatos seleccionados*.
- $solucion()$ es la *función solución*.
- $seleccionar()$ es la *función de selección*.
- $factible()$ es la *función factible*.

Estos conjuntos y funciones habrá que particularizarlos para cada problema concreto.
La descripción del algoritmo por pasos es la siguiente:

1. El conjunto inicial de candidatos ya escogidos está vacío.
2. La función de selección elige el mejor candidato de los aún no escogidos.
3. Si el conjunto resultante no es completable o factible, se rechaza el candidato y no se vuelve a considerar.
4. Si el conjunto resultante es completable, se incorpora el candidato al conjunto de candidatos escogidos.
5. Se comprueba si el conjunto resultante es una solución al problema y si quedan más candidatos.
6. En el caso de que no sea una solución y queden más candidatos que examinar se vuelve al paso 2.

Si nos fijamos en el esquema, vemos que como mucho se examina una vez cada candidato y se decide si se selecciona o se rechaza. La función de selección intentará seleccionar al mejor de los candidatos restantes, y por lo tanto está relacionada con la función objetivo. Por ejemplo, si se trata de minimizar el coste, la función de selección escogerá al candidato más barato de los restantes. Si el algoritmo es correcto, la primera solución encontrada es óptima.

La principal característica que distingue al esquema voraz de otros esquemas es que nunca deshace una decisión ya tomada. Cuando se incorpora un candidato a la solución permanece hasta el final, y cuando se rechaza un candidato no se vuelve a tener en cuenta.

En general, los algoritmos voraces pueden parecer sencillos y, al no reconsiderar decisiones, suelen resultar eficientes. Sin embargo, llevan asociada una *demonstración de*

optimalidad, es decir, una demostración de que efectivamente la función de selección utilizada lleva a una solución óptima que, en ocasiones, es compleja.

Para ilustrar este esquema vamos a utilizar uno de los problemas más conocidos que en una de sus modalidades se pueden resolver con un algoritmo voraz: la devolución de cambio de monedas.

Supongamos que disponemos de un conjunto finito de tipos de moneda $T = \{m^0, m^1, m^2, \dots, m^n\}$, es decir n tipos de moneda, que $m > 1$ y que $n > 0$, por lo que los valores de las monedas son potencias consecutivas de m . El problema consiste en pagar una cantidad $C > 0$ utilizando un número mínimo de monedas y suponiendo que la disponibilidad de cada tipo de moneda es ilimitada. Probablemente la estrategia voraz que primero se nos ocurre es seleccionar los tipos de moneda de mayor a menor, cogiendo tantas unidades como sea posible de cada tipo hasta llegar a C . Efectivamente, en este caso, esta estrategia voraz nos llevaría a obtener una solución óptima. El algoritmo que resuelve este problema es el siguiente:

```

tipo VectorNat = matriz[1..n] de natural
fun MonedasCambio(T: VectorNat, C: natural): VectorNat
    var
        solucion: VectorNat
    fvar
        para i  $\leftarrow$  1 hasta n hacer
            solucion[i]  $\leftarrow$  0
        fpara
        canRestante  $\leftarrow$  C
        i  $\leftarrow$  n
        mientras canRestante  $\neq$  0  $\wedge$  i  $\geq$  1 hacer
            solucion[i]  $\leftarrow$  canRestante div T[i]
            canRestante  $\leftarrow$  canRestante mod T[i]
            i  $\leftarrow$  i - 1
        fmientras
        dev solucion[]
ffun
```

El array $T[]$ almacena los n tipos de moneda $\{m^0, m^1, m^2, \dots, m^n\}$ y el array $solucion[]$ almacena en la posición i -ésima la cantidad de monedas del tipo i necesarias para devolver el cambio.

Supongamos que disponemos de los tipos de moneda $T = \{1, 2, 4, 8\}$ que satisfacen las restricciones descritas anteriormente y que queremos pagar la cantidad 61. El algoritmo primero seleccionaría la moneda mayor, el 8. Al dividir 61 entre 8 da el entero 7 y queda como cantidad restante 5. Seleccionaría el siguiente tipo de moneda mayor, el 4. Al dividir 5 entre 4 da el entero 1 y queda como cantidad restante 1. A continuación

se seleccionaría el siguiente tipo de moneda mayor, el 2, y al dividir 1 entre 2 da el entero 0 y queda como cantidad restante 1. Finalmente se seleccionaría el último tipo de moneda, el 1 y al dividir 1 entre 1 da el entero 1 y queda como cantidad restante 0. Ambas condiciones de salida del bucle serían falsas por lo que el algoritmo terminaría, quedando como resultado: 7 monedas de 8, 1 moneda de 4 y 1 moneda de 1.

Podemos identificar los elementos del esquema voraz en este algoritmo:

- Resolución de un problema de forma óptima: descomponer una cantidad en un número mínimo de monedas.
- *Conjunto inicial de candidatos*: los distintos tipos de moneda.
- *Conjunto de candidatos seleccionados y conjunto de candidatos rechazados*: están representados en el array *solucion[]* desde las posiciones n -ésima a la i -ésima. Los candidatos seleccionados tendrán en su posición del array un valor mayor que 0, y los rechazados tendrán un valor igual a 0.
- Las funciones *solución* y *factible* están implícitas en las condiciones de finalización del bucle.
- La *función de selección* se traduce en la selección de los elementos del array *T[]* según el orden: $n, n - 1, n - 2, \dots, 1$.
- La *función objetivo* no aparece explícitamente en el algoritmo.

La demostración de optimalidad se apoya en la siguiente propiedad general de los números naturales: si m es un número natural mayor que 1, todo número natural c puede expresarse de forma única como:

$$c = v_0m^0 + v_1m^1 + v_2m^2 + \cdots + v_nm^n \quad (3.1)$$

siendo $0 \leq v_i < m$ para todo $0 \leq i \leq n$ y siendo n el menor natural tal que $c < m^{n+1}$.

Como el algoritmo lo que hace es calcular los v_i asociados a los m_i , es decir, el número de monedas de cada tipo en que se descompone la cantidad c , hay que demostrar que la descomposición que se obtiene con los v_i es óptima. Esto supone que si:

$$c = s_0m^0 + s_1m^1 + s_2m^2 + \cdots + s_pm^p$$

es una descomposición distinta de la misma cantidad c , entonces:

$$\sum_{i=0}^n v_i < \sum_{i=0}^p s_i$$

La demostración se hará para el caso más sencillo $m = 2$ ya que el caso general es similar. Sea la descomposición obtenida por el algoritmo voraz la siguiente:

$$c = v_02^0 + v_12^1 + v_22^2 + \cdots + v_n2^n = v_0 + 2v_1 + 2^2v_2 + \cdots + 2^nv_n \quad (3.2)$$

tenemos que $c < 2^{n+1}$ y que $0 \leq v_i < 2$ por lo que los coeficientes v_i tendrán el valor 0 o 1. Si la siguiente descomposición es distinta:

$$c = s_0 + 2s_1 + 2^2s_2 + \cdots + 2^ps_p$$

como tenemos que $c < 2^{n+1}$, implica que $p \leq n$ (porque si $p > n$ entonces se daría que $m^p > c$). Con el fin de tener n términos en cada descomposición se definen $s_{p+1} = s_{p+2} = \cdots = s_n = 0$. Se trata entonces de demostrar que:

$$v_0 + v_1 + v_2 + \cdots + v_n < s_0 + s_1 + s_2 + \cdots + s_n$$

Como ambas descomposiciones son distintas, sea k el primer índice tal que $v_k \neq s_k$. Supongamos sin pérdida de generalidad que $k = 0$ (en caso de que $k \neq 0$ se podrían eliminar de la desigualdad los términos iguales y dividir por la potencia de 2 correspondiente), como $v_0 \neq s_0$ veamos cómo es v_0 con respecto a s_0 .

- Si c es par, entonces $v_0 = 0$. Como $s_0 \geq 0$ y $v_0 \neq s_0$ entonces $v_0 < s_0$.
- Si c es impar, entonces $v_0 = 1$ y por lo tanto $s_0 \geq 1$. Como también sabemos que $v_0 \neq s_0$, entonces $s_0 > 1$ y en este caso también $v_0 < s_0$.

Por lo anterior, $s_0 - v_0 > 0$. Además, esta cantidad debe ser par ya que siendo $m = 2$ la cantidad $c - v_0$ es par por la expresión 3.2. Al ser par, siempre se podrá mejorar la descomposición (s_0, s_1, \dots, s_n) cambiando $s_0 - v_0$ monedas de 1 unidad por $(s_0 - v_0)/2$ monedas de 2 unidades, obteniendo así:

$$s_0 + s_1 + s_2 + \cdots + s_n > v_0 + \left(s_1 + \frac{s_0 - v_0}{2}\right) + s_2 + \cdots + s_n \quad (3.3)$$

Utilizando el razonamiento anterior se ha obtenido una nueva descomposición mejor y manteniendo:

$$v_0 + \left(s_1 + \frac{s_0 - v_0}{2}\right)2 + s_22^2 + \cdots + s_n2^n = c = v_0 + v_12 + \cdots + v_n2^n$$

Si aplicamos sucesivamente el razonamiento anterior a la nueva descomposición se puede ir viendo que $s_i \geq v_i$ para todo $0 \leq i \leq n-1$ y así ir obteniendo nuevas descomposiciones, cada una mejor que la anterior, hasta llegar en el último paso a la siguiente descomposición:

$$v_0 + v_12 + v_22^2 + \cdots + v_{n-1}2^{n-1} + \left(s_n + \frac{s_{n-1} - \text{acum}_{n-1}}{2}\right)2^n = c \quad (3.4)$$

en la que se han ido acumulando las diferencias en el último término y que además verifica que:

$$v_0 + v_1 + \cdots + v_i + \left(s_{i+1} + \frac{s_i - \text{acum}_i}{2}\right) + \cdots + s_n \geq v_0 + v_1 + \cdots + v_n, \quad 0 \leq i \leq n-1$$

Si se verifica 3.4 por la unicidad de la descomposición a la que hacía referencia la propiedad 3.1 se ha de cumplir que:

$$s_n + \frac{s_{n-1} - \text{acum}_{n-1}}{2} = v_n$$

que junto a las desigualdades 3.3 hace que quede demostrada la afirmación. El razonamiento es igual para $m > 2$.

Este algoritmo no siempre obtiene el número mínimo de monedas cuando no se cumplen las restricciones del enunciado. Por ejemplo, si tuviéramos sólo monedas de valores 1, 15 y 25 y quisieramos calcular el mejor cambio para la cantidad 35, este algoritmo nos indicaría: una moneda de 25 y diez de 1, en total 11 monedas. Mientras que la solución óptima sería tres monedas de 10 y cinco de 1, que hacen un total de 8 monedas. Si el sistema monetario no dispone de monedas de 1, es posible que no se pueda realizar el cambio con este algoritmo. Por ejemplo con monedas de valores 4, 10 y 25 no se podría cambiar la cantidad 41.

En cuanto al coste del algoritmo *MonedasCambio*, los dos bucles consecutivos se ejecutan tantas veces como el número de tipos de moneda, m , y ambos contienen instrucciones de coste constante, por lo que el coste global está en $O(m)$.

3.1.1 Algoritmos voraces como procedimientos heurísticos

En la bibliografía, algunos autores, como por ejemplo [AHU98], también denominan voraces a algoritmos que aunque no obtienen una solución óptima, utilizan una estrategia voraz y pueden obtener una solución cercana a la óptima en la mayoría de los casos. Estas soluciones se denominan soluciones subóptimas. Esto puede ser útil a la hora de tratar problemas cuya única forma de llegar a una solución óptima sea sumamente costosa, y el hecho de encontrar una solución buena a un coste razonable es la única opción aplicable. Nosotros en este capítulo nos centraremos sólo en los algoritmos voraces que obtienen soluciones óptimas.

3.2 Algoritmos voraces con grafos

Hay dos problemas de grafos muy conocidos que se solucionan con algoritmos voraces: hallar un árbol de recubrimiento de distancia o coste mínimo y calcular el camino de coste mínimo entre un nodo y los demás.

En el apartado 2.1.5 del capítulo 2 se ha visto lo que es un árbol de recubrimiento asociado a un grafo. Un *árbol de recubrimiento mínimo* o de coste mínimo es aquel cuya suma de los pesos de sus aristas es la mínima posible. Sea $G = \langle N, A \rangle$ un grafo conexo y no dirigido cuyas aristas tienen asignados pesos no negativos. Encontrar un árbol de recubrimiento mínimo consiste en seleccionar un subconjunto AR de las aristas de G , tal que si utilizamos sólo dicho subconjunto de aristas todos los nodos están conectados y la suma de los pesos de dichas aristas es mínima.

Tal y como se veía en el apartado 2.1.5, si el grafo es conexo se le puede asociar al menos un árbol de recubrimiento. Si el grafo tiene algunas aristas de igual peso, puede darse el caso de que haya varias soluciones con igual peso, pero con distintas aristas. Si el grafo tiene algunas aristas de longitud 0 también podrá haber más de una solución con igual peso, pero distinto número de aristas. En este caso se trataría de seleccionar la solución con menor número de aristas. Nótese que un grafo conexo con n nodos debe tener al menos $n - 1$ aristas, por lo que un árbol de recubrimiento de coste mínimo deberá tener $n - 1$ aristas. Este problema se suele aplicar para asegurar la conectividad en redes de servidores, de ciudades, etc. Por ejemplo, se puede aplicar para garantizar la conexión entre una serie de ciudades de forma que el coste de la misma sea mínimo.

El problema de calcular el camino de coste mínimo entre un nodo y los demás se formula de la siguiente manera: sea $G = \langle N, A \rangle$ un grafo dirigido, con pesos mayores o iguales que cero en sus aristas, en el que todos sus nodos son accesibles desde uno concreto considerado como origen. El problema consiste en determinar la longitud del camino mínimo que va desde el nodo origen a cada uno de los demás nodos del grafo. En cuanto a su aplicación, los nodos del grafo pueden representar ciudades y las aristas las carreteras que las unen junto con sus distancias, de tal manera que permitiría calcular la ruta más corta entre una ciudad y las demás o entre dos ciudades, si se detiene el algoritmo una vez que el nodo destino ha sido examinado. Este problema también permite modelar tareas en enrutamiento de redes.

En los apartados siguientes estudiaremos los algoritmos más conocidos para solucionar estos dos problemas.

3.2.1 Árboles de recubrimiento mínimo: algoritmo de Prim

Este algoritmo fue propuesto por primera vez en 1930 por el matemático checo Vojtech Jarník, posteriormente y de forma independiente fue propuesto en 1957 por Robert C. Prim, siendo redescubierto por Edsger Dijkstra in 1959. El conocido como algoritmo de Prim selecciona arbitrariamente un nodo del grafo como raíz del árbol AR . En cada paso se añade al árbol una arista de coste mínimo (u, v) tal que $AR \cup \{(u, v)\}$ sea también un árbol. El algoritmo continúa hasta que AR contiene $n - 1$ aristas. Nótese que la arista (u, v) que se selecciona en cada paso es tal que o bien u o v está en AR .

Para la descripción de alto nivel del algoritmo de Prim se van a utilizar dos conjuntos: un conjunto con los nodos seleccionados NA y el conjunto de aristas del árbol de recubrimiento mínimo AR .

fun Prim ($G = \langle N, A \rangle$: grafo): conjunto de aristas

$AR \leftarrow \emptyset$

$NA \leftarrow \{\text{un nodo cualquiera de } N\}$

mientras $NA \neq N$ **hacer**

 Buscar $\{u, v\}$ de coste mínimo tal que $u \in NA$ y $v \in N \setminus NA$

$AR \leftarrow AR \cup \{(u, v)\}$

```

NA ← NA ∪ {v}
fmientras
  dev AR
ffun

```

Veamos si la función de selección del algoritmo de Prim conduce a una solución óptima.

Demostración de optimalidad

La demostración de optimalidad se realiza por inducción y utiliza el concepto de *conjunto prometedor*. Un conjunto de aristas factible, es decir, que no contiene ningún ciclo, es un conjunto prometedor si se puede extender para producir una solución óptima. El conjunto vacío es prometedor, ya que al ser G conexo siempre existe una solución óptima. Además, si un conjunto prometedor es ya una solución, y por lo tanto no requiere extensión, esa solución debe ser óptima. Se dice que una arista sale de un conjunto de nodos si esa arista tiene sólo un extremo en dicho conjunto. Esto supone que si una arista no tiene ninguno de sus extremos en un conjunto de nodos o bien tiene ambos, no sale de él.

Lema 3.2.1 *Sea $G = \langle N, A \rangle$ un grafo conexo, no dirigido, con pesos mayores o iguales que cero en sus aristas. Sea $NA \subset N$ un subconjunto estricto de los nodos de G . Sea $AR \subseteq A$ un conjunto prometedor de aristas tal que no haya ninguna arista de AR que sale de algún nodo de NA . Sea (u, v) la arista de peso menor que salga de NA (o una de las de peso menor si hay más de una con igual peso), entonces $AR \cup \{(u, v)\}$ es un conjunto prometedor.*

Veamos la demostración de este lema. Sea ARM un árbol de recubrimiento mínimo de G tal que $AR \subseteq ARM$. Nótese que ARM tiene que existir ya que AR es prometedor por hipótesis. Si la arista $(u, v) \in ARM$ no hay nada que demostrar. Si no, al añadir (u, v) a ARM se crea un ciclo. En este ciclo, como (u, v) sale de NA existe necesariamente al menos otra arista (x, y) que también sale de NA o el ciclo no se cerraría. Si eliminamos (x, y) el ciclo desaparece y obtenemos un nuevo árbol ARM' de G . Como la longitud de (u, v) , por definición, no es mayor que la longitud de (x, y) , la longitud total de las aristas de ARM' no sobrepasa la longitud total de las aristas de ARM . Por lo tanto, ARM' es también un árbol de recubrimiento mínimo de G y contiene a (u, v) . Como la arista que se ha eliminado sale de NA , no podría haber sido una arista de AR y se cumple que $AR \subseteq ARM'$.

La demostración de optimalidad que garantiza que el algoritmo de Prim halla un árbol de recubrimiento mínimo se realiza por inducción y parte del lema anterior:

Base: el conjunto vacío es prometedor.

Paso inductivo: suponemos que AR es un conjunto de aristas prometedor antes de que el algoritmo añada una nueva arista (u, v) ; NA es un subconjunto estricto de N

ya que el algoritmo termina una vez que $NA = N$; la arista (u, v) es una de las de menor peso de las que salen de NA . Entonces podemos utilizar el lema 3.2.1 ya que se cumplen sus condiciones y por lo tanto $AR \cup \{(u, v)\}$ también es prometedor.

Como AR es prometedor en todos los pasos del algoritmo, cuando el algoritmo se detenga AR contendrá una solución óptima.

Descripción detallada del algoritmo de Prim

Para una descripción más detallada del algoritmo es necesario concretar las estructuras de datos que permiten acceder a las aristas de coste mínimo no incluidas en el árbol de recubrimiento, y que permitirán conectar algún nodo del árbol con otro que aún no está en el árbol. Supongamos que los nodos están numerados de 1 a n , por lo que $N = \{1, 2, \dots, n\}$, y que el nodo 1 es el que consideramos la raíz del árbol. Para todo nodo $i \in N \setminus NA$, sea el array $nodoMinimo[i]$ el que indica el nodo de NA más próximo o con menos coste al i -ésimo, y sea $costeMinimo[i]$ el coste desde el nodo i a dicho nodo. Cuando un nodo esté en NA su valor de $costeMinimo[i]$ tendrá un valor fuera del rango de los costes, por ejemplo -1, lo que indicará que ya está en NA . El elemento $nodoMinimo[1]$ no se utiliza en el algoritmo. El conjunto de nodos ya incluidos en el árbol, NA , tampoco se utiliza.

```

tipo VectorNat = matriz[0..n] de natural
tipo VectorEnt = matriz[0..n] de entero
fun PrimDetallado ( $G = \langle N, A \rangle$ : grafo): conjunto de aristas
    var
        nodoMinimo: VectorNat
        costeMinimo: VectorEnt
        AR: conjunto de aristas
    fvar
         $AR \leftarrow \emptyset$ 
         $costeMinimo[1] \leftarrow -1$ 
        para  $i \leftarrow 2$  hasta  $n$  hacer
             $nodoMinimo[i] \leftarrow 1$ 
             $costeMinimo[i] \leftarrow \text{Distancia}(1,i)$ 
        fpara
        para  $i \leftarrow 1$  hasta  $n-1$  hacer
             $\min \leftarrow \infty$ 
            para  $j \leftarrow 2$  hasta  $n$  hacer
                si  $0 \leq costeMinimo[j] \wedge costeMinimo[j] < \min$  entonces
                     $\min \leftarrow costeMinimo[j]$ 
                     $nodo \leftarrow j$ 
            fsi
        
```

```

fpara
AR ← AR ∪ {(nodoMinimo[nodo], nodo)}
costeMinimo[nodo] ← -1
para j ← 2 hasta n hacer
    si Distancia(j,nodo) < costeMinimo[j] ∧ costeMinimo[j] ≠ -1 entonces
        costeMinimo[j] ← Distancia(j,nodo)
        nodoMinimo[j] ← nodo
    fsi
fpara
fpara
dev AR
ffun

```

La función *Distancia()* devolverá la distancia o coste entre los dos nodos que son sus argumentos. Si existe una arista entre dichos nodos devolverá la etiqueta o peso asociado. En el caso de que no exista una arista devolverá un valor representativo o suficientemente grande, por ejemplo ∞ .

Veamos cómo se comporta el algoritmo con el grafo de la figura 3.1 cuando se toma como nodo raíz el nodo 1. Para ello vamos a hacer la traza de la descripción más detallada del algoritmo de Prim:

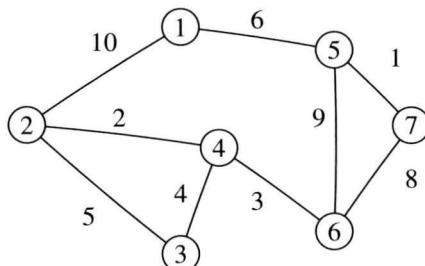


Figura 3.1: Grafo conexo no dirigido con pesos

| Paso | nodoMinimo[] | costeMinimo | min | nodo | AR |
|--------|---------------|----------------------|--------|-------|------------|
| Inicio | _ 1 1 1 1 1 1 | -1 10 ∞ ∞ 6 ∞ ∞ | | | ∅ |
| i=1 | | | ∞ | | |
| j=2..n | | -1 10 ∞ ∞ -1 ∞ ∞ | 10,6 | 2,5 | AR ∪ {1,5} |
| j=2..n | _ 1 1 1 1 5 5 | -1 10 ∞ ∞ -1 9 1 | | | |
| i=2 | | | ∞ | | |
| j=2..n | | -1 10 ∞ ∞ -1 9 -1 | 10,9,1 | 2,6,7 | AR ∪ {5,7} |
| j=2..n | _ 1 1 1 1 7 5 | -1 10 ∞ ∞ -1 8 -1 | | | |
| i=3 | | | ∞ | | |
| j=2..n | | -1 10 ∞ ∞ -1 -1 -1 | 10,8 | 2,6 | AR ∪ {7,6} |
| j=2..n | _ 1 1 6 1 7 5 | -1 10 ∞ 3 -1 -1 -1 | | | |
| i=4 | | | ∞ | | |
| j=2..n | | -1 10 ∞ -1 -1 -1 -1 | 10,3 | 2,4 | AR ∪ {6,4} |
| j=2..n | _ 4 4 6 1 7 5 | -1 2 4 -1 -1 -1 -1 | | | |
| i=5 | | | ∞ | | |
| j=2..n | | -1 -1 4 -1 -1 -1 -1 | 2 | 2 | AR ∪ {4,2} |
| j=2..n | _ 4 4 6 1 7 5 | -1 -1 4 -1 -1 -1 -1 | | | |
| i=6 | | | ∞ | | |
| j=2..n | | -1 -1 -1 -1 -1 -1 -1 | 4 | 3 | AR ∪ {4,3} |
| j=2..n | _ 4 4 6 1 7 5 | -1 -1 -1 -1 -1 -1 -1 | | | |

Tras la ejecución del algoritmo, el árbol de recubrimiento mínimo resultante es el que se muestra en la figura 3.2. Las aristas con líneas continuas pertenecen al árbol AR , mientras que las líneas discontinuas representan las aristas de $A \setminus AR$.

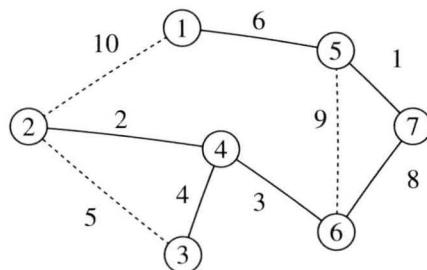


Figura 3.2: Árbol de recubrimiento mínimo

Coste

El bucle principal se ejecuta $n - 1$ veces y dentro de este bucle hay una secuencia de dos bucles que se ejecutan $n - 2$ veces, por lo que el coste está en $O(n^2)$. Si el grafo se implementa mediante una matriz de adyacencia el coste está en $O(n^2)$ ya que es independiente del número de aristas. Si el grafo se implementa mediante listas de adyacencia y se utiliza un montículo para representar los candidatos pendientes, el coste está en

$O(n \log n + a \log n) = O(a \log n)$, siendo a el número de aristas, ya que al ser el grafo conexo $n - 1 \leq a \leq n^2$. Esto se debe a que la inicialización del montículo está en $O(n)$ (ver apartado del tema montículos) y eliminar la raíz del montículo está en $O(\log n)$. El uso de la lista de adyacencia en combinación con el montículo es más apropiado sólo si el grafo es disperso (número de aristas cercano a n), ya que si el grafo es denso (número de aristas cercano a n^2) el coste pasa a ser $O(n^2 \log n)$ que es peor que $O(n^2)$.

3.2.2 Árboles de recubrimiento mínimo: algoritmo de Kruskal

Este algoritmo fue propuesto por Joseph Kruskal en 1956. Partiendo del conjunto AR (conjunto de aristas del árbol de recubrimiento mínimo) vacío, en cada paso selecciona la arista más corta o con menos peso que todavía no haya sido añadida a AR o rechazada. Inicialmente, AR está vacío y cada nodo de G forma una componente conexa. En los pasos intermedios, el grafo parcial formado por los nodos de G y las aristas de AR consta de varias componentes conexas. Al final del algoritmo sólo queda una componente conexa, que es el árbol de recubrimiento mínimo de G .

Cuando en cada paso se selecciona la arista más corta que todavía no ha sido evaluada, se determina si une dos nodos pertenecientes a dos componentes conexas distintas. En caso afirmativo la arista se añade a AR , por lo que las dos componentes conexas ahora forman una. En caso contrario, esa arista se rechaza ya que forma un ciclo al unir dos nodos de la misma componente conexa.

Para la descripción de alto nivel del algoritmo de Kruskal se van a utilizar tantos conjuntos como componentes conexas haya, cuyo número inicialmente es igual al número de nodos. Hay dos operaciones relevantes que son *buscarComponenteConexa(x)* y *fusionar(C1, C2)* que se ocupan, respectivamente, de determinar en qué componente conexa está el nodo x , y de unir dos componentes conexas en una.

```

fun Kruskal ( $G = \langle N, A \rangle$ ): grafo: conjunto de aristas
  var
    AR: conjunto de aristas
  fvar
    Ordenar(A) {Ordena  $A$  en pesos crecientes}
    n  $\leftarrow$  no nodos de N
    AR  $\leftarrow$   $\emptyset$ 
    Iniciar n conjuntos, uno con cada nodo de N
    mientras AR no tenga  $n-1$  aristas hacer
      seleccionar {u,v} mínima
      comU  $\leftarrow$  buscarComponenteConexa(u)
      comV  $\leftarrow$  buscarComponenteConexa(v)
      si comU  $\neq$  comV entonces
        fusionar(comU,comV)
        AR  $\leftarrow$  AR  $\cup$  {(u,v)}
    
```

```

fsi
fmientras
  dev AR
ffun

```

Veamos ahora cómo se comporta el algoritmo con el grafo de la figura 3.1 a través de su traza:

| Paso | arista | Componentes conexas | AR |
|--------|-----------|---------------------------------------|-------------------|
| Inicio | | $\{1\}\{2\}\{3\}\{4\}\{5\}\{6\}\{7\}$ | \emptyset |
| 1 | $\{5,7\}$ | $\{1\}\{2\}\{3\}\{4\}\{5,7\}\{6\}$ | $AR \cup \{5,7\}$ |
| 2 | $\{2,4\}$ | $\{1\}\{2,4\}\{3\}\{5,7\}\{6\}$ | $AR \cup \{2,4\}$ |
| 3 | $\{4,6\}$ | $\{1\}\{2,4,6\}\{3\}\{5,7\}$ | $AR \cup \{4,6\}$ |
| 4 | $\{3,4\}$ | $\{1\}\{2,4,6,3\}\{5,7\}$ | $AR \cup \{3,4\}$ |
| 5 | $\{2,3\}$ | se rechaza | |
| 6 | $\{1,5\}$ | $\{1,5,7\}\{2,4,6,3\}$ | $AR \cup \{1,5\}$ |
| 7 | $\{6,7\}$ | $\{1,5,7,2,4,6,3\}$ | $AR \cup \{6,7\}$ |

El árbol de recubrimiento mínimo resultante es también el de la figura 3.2.

Demostración de optimalidad

La demostración de optimalidad del algoritmo de Kruskal se realiza por inducción. Se trata de demostrar de que si AR es prometedor seguirá siéndolo en cualquier fase del algoritmo cuando se le añada una arista:

Base: el conjunto vacío es prometedor.

Paso inductivo: suponemos que AR es un conjunto de aristas prometedor antes de que el algoritmo añada una nueva arista (u, v) . En ese momento, el nodo u se encuentra en una componente conexa y el nodo v en otra distinta. Sea C_i el conjunto de nodos de la componente que contiene a u . Tenemos que:

- El conjunto C_i es un subconjunto estricto del conjunto de nodos N de G , ya que al menos no incluye a v .
- AR es un conjunto prometedor tal que ninguna arista de AR sale de C_i .
- (u, v) es una de las aristas más cortas que salen de C_i , ya que las que sean estrictamente más cortas ya se han examinado, y se han añadido a AR o se han rechazado.

Entonces podemos utilizar el lema 3.2.1 ya que se cumplen sus condiciones y por lo tanto $AR \cup \{(u, v)\}$ también es prometedor.

Como AR es prometedor en todos los pasos del algoritmo, cuando el algoritmo se detenga AR contendrá una solución óptima.

Coste

En cuanto al coste, la ordenación de las aristas está en $O(a \log a)$, siendo a el número de aristas de G , que equivale a $O(a \log n)$ ya que $n - 1 \leq a \leq n(n - 1)/2$ y $O(a \log n) \simeq O(a \log n^2)$. Iniciar n conjuntos disjuntos está en $O(n)$. El bucle principal, en el caso peor de que se examinen todas las aristas se ejecutará a veces, se ejecutará $2a$ veces `buscarComponenteConexa()` y $n - 1$ veces `fusionar()`. Por lo que el coste total del algoritmo puede considerarse el coste de la ordenación e inicialización que está en $O(a \log n)$. En este caso, utilizar un montículo para almacenar las aristas pendientes no modifica el análisis en el caso peor, aunque resulta ventajoso si el árbol de recubrimiento se encuentra cuando queden muchas aristas sin seleccionar ya que no se emplea tiempo en su ordenación.

Comparando los dos algoritmos, el de Prim y el de Kruskal, si el grafo es denso será preferible el algoritmo de Prim ya que su coste es $O(n^2)$, pero si el grafo es disperso será más adecuado el algoritmo de Kruskal.

3.2.3 Camino de coste mínimo: algoritmo de Dijkstra

Sea $G = \langle N, A \rangle$ un grafo dirigido, con pesos mayores o iguales que cero en sus aristas, en el que todos sus nodos son accesibles desde uno concreto considerado como origen. El algoritmo de Dijkstra [Dij59] determina la longitud del camino de coste, peso o distancia mínima que va desde el nodo origen a cada uno de los demás nodos del grafo. También se puede utilizar para calcular el camino de coste o peso mínimo entre el origen y un único nodo del grafo. Para ello habría que detener el algoritmo una vez que el camino hasta dicho nodo ya se ha calculado.

Este algoritmo utiliza dos conjuntos de nodos, S y C . S contiene los nodos ya seleccionados y cuya distancia mínima al origen ya se conoce y C contiene los demás nodos, $C = N \setminus S$, aquellos cuya distancia mínima al origen no se conoce todavía. Al inicio del algoritmo S sólo contiene el nodo origen y cuando finaliza el algoritmo contiene todos los nodos del grafo y además se conocen las longitudes mínimas desde el origen a cada uno de ellos. La función de selección elegirá en cada paso el nodo de C cuya distancia al origen sea mínima.

El algoritmo de Dijkstra utiliza la noción de *camino especial*. Un camino desde el nodo origen hasta otro nodo es especial si todos los nodos intermedios del camino pertenecen a S , es decir, se conoce el camino mínimo desde el origen a cada uno de ellos. Hará falta un array, `especial[]`, que en cada paso del algoritmo contendrá la longitud del camino especial más corto (si el nodo está en S), o el camino más corto conocido (si el nodo está en C) que va desde el origen hasta cada nodo del grafo. Cuando se va a añadir un nodo a S , el camino especial más corto hasta ese nodo es también el más corto de todos los caminos posibles hasta él. Cuando finaliza el algoritmo todos los nodos están en S y por lo tanto todos los caminos desde el origen son caminos especiales.

Las longitudes o distancias mínimas que se esperan calcular con el algoritmo estarán almacenadas en el array *especial*[].

Supongamos que los nodos están numerados de 1 a n , por lo que $N = \{1, 2, \dots, n\}$, y que el nodo 1 es el nodo origen. Al igual que en algoritmo de Prim, suponemos que la función *Distancia()* devolverá la distancia o coste entre los dos nodos que son sus argumentos. Si existe una arista entre dichos nodos devolverá la etiqueta o peso asociado, en caso de que no exista una arista devolverá un valor representativo o suficientemente grande, por ejemplo ∞ . El algoritmo de Dijkstra quedaría:

```
tipo VectorNat = matriz[0..n] de natural
fun Dijkstra ( $G = \langle N, A \rangle$ : grafo): VectorNat
    var
        especial: VectorNat
        C: conjunto de nodos
    fvar
        C = {2, 3, ..., n}
    para i  $\leftarrow$  2 hasta n hacer
        especial[i]  $\leftarrow$  Distancia(1,i)
    fpara
    mientras C contenga más de 1 nodo hacer
        v  $\leftarrow$  nodo  $\in$  C que minimiza especial[v]
        C  $\leftarrow$  C \{v\}
        para cada w  $\in$  C hacer
            especial[w]  $\leftarrow$  min(especial[w], especial[v] + Distancia(v,w))
        fpara
    fmientras
    dev especial[]
ffun
```

El bucle **mientras** se ejecuta $n - 2$ veces ya que cuando sólo queda un nodo en C no va a haber más modificaciones en el array *especial*[] . Si además de calcular el coste del camino mínimo desde el origen se quiere saber por dónde pasan los caminos, es necesario utilizar un array adicional, *predecesor*[2..n], siendo *predecesor*[i] el identificador del nodo que precede al nodo i-ésimo en el camino más corto desde el origen. Añadiendo esta nueva funcionalidad el algoritmo quedaría:

```
tipo VectorNat = matriz[0..n] de natural
fun Dijkstra ( $G = \langle N, A \rangle$ : grafo): VectorNat, VectorNat
    var
        especial, predecesor: VectorNat
        C: conjunto de nodos
```

fvar

$$C = \{2, 3, \dots, n\}$$

para $i \leftarrow 2$ **hasta** n **hacer** especial[i] \leftarrow Distancia(1,i) predecesor[i] $\leftarrow 1$ **fpara****mientras** C contenga más de 1 nodo **hacer** $v \leftarrow$ nodo $\in C$ que minimiza especial[v] $C \leftarrow C \setminus \{v\}$ **para** cada $w \in C$ **hacer** **si** especial[w] $>$ especial[v] + Distancia(v,w) **entonces** especial[w] \leftarrow especial[v] + Distancia(v,w) predecesor[w] $\leftarrow v$ **fsi****fpara****fmientras****dev** especial[],predecesor[]**ffun**

Veamos cómo se comporta el algoritmo con el grafo de la figura 3.3 cuando se toma como nodo origen el nodo 1.

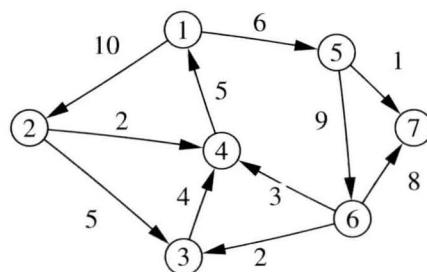


Figura 3.3: Grafo conexo dirigido con pesos

| Paso | v, w | C | especial[] | predecesor[] |
|--------|------------------------|---------------|-----------------|--------------|
| Inicio | | {2,3,4,5,6,7} | 10 ∞ ∞ 6 ∞ ∞ | 1 1 1 1 1 1 |
| 1 | v = 5 w = 2,3,4,6,7 | {2,3,4,6,7} | 10 ∞ ∞ 6 15 7 | 1 1 1 1 5 5 |
| 2 | v = 7 w = 2,3,4,6 | {2,3,4,6} | 10 ∞ ∞ 6 15 7 | 1 1 1 1 5 5 |
| 3 | v = 2 w = 3,4,6 | {3,4,6} | 10 15 12 6 15 7 | 1 2 2 1 5 5 |
| 4 | v = 4 w = 3,6 | {3,6} | 10 15 12 6 15 7 | 1 2 2 1 5 5 |
| 5 | v = 3 w = 6 | {6} | 10 15 12 6 15 7 | 1 2 2 1 5 5 |

Demostración de optimalidad

La demostración de que el algoritmo de Dijkstra calcula los caminos de menor coste o longitud desde un nodo tomado como origen a los demás nodos del grafo se realiza por inducción. Se trata de demostrar que:

- Si un nodo $i \neq 1$ está en S , entonces $\text{especial}[i]$ almacena la longitud del camino más corto desde el origen, 1, hasta el nodo i .
- Si un nodo i no está en S , entonces $\text{especial}[i]$ almacena la longitud del camino especial más corto desde el origen, 1, hasta el nodo i .

Además, por hipótesis estos dos puntos se cumplen inmediatamente antes de añadir un nodo v al conjunto S .

Base: inicialmente sólo el nodo 1 esta en S , por lo que el punto 1 está demostrado. Para el resto de los nodos el único camino especial posible es el camino directo al nodo 1, cuya distancia es la que le asigna el algoritmo a $\text{especial}[i]$, por lo que el punto 2 también está demostrado.

Paso inductivo: como los nodos que ya están en S no se vuelven a examinar, el punto 1 se sigue cumpliendo antes de añadir un nodo v a S . Antes de poder añadir el nodo v a S hay que comprobar que $\text{especial}[v]$ almacene la longitud del camino más corto o de menor coste desde el origen hasta v . Por hipótesis, $\text{especial}[v]$ almacena la longitud del camino especial más corto, por lo que hay que verificar que dicho camino no pase por ningún nodo que no pertenece a S . Supongamos que en el camino más corto desde el origen a v hay uno o más nodos, que no son v , que no pertenecen a S . Sea x el primer nodo de este tipo. Como x no está en S , $\text{especial}[x]$ almacena la longitud del camino especial desde el origen hasta x . Como el algoritmo ha seleccionado a v antes que a x , $\text{especial}[x]$ no es menor que $\text{especial}[v]$. Por lo tanto, la distancia total hasta v a través de x es

como mínimo $\text{especial}[v]$, y el camino a través de x no puede ser más corto que el camino especial que lleva a v . Por lo tanto, cuando se añade v a S se cumple el punto 1.

Con respecto al punto 2, consideremos un nodo u que no sea v y que no pertenece a S . Cuando se añade v a S puede darse una de dos posibles situaciones: (1) $\text{especial}[u]$ no cambia porque no se encuentra un camino más corto a través de v o, (2) $\text{especial}[u]$ si cambia porque se encuentra un camino más corto a través de v y quizás de algún otro u otros nodos de S . En este caso (2), sea x el último nodo de S visitado antes de llegar a u . La longitud de ese camino será $\text{especial}[x] + \text{Distancia}(x, u)$. Se podría pensar que para calcular el nuevo valor de $\text{especial}[u]$ habría que comparar el valor anterior de $\text{especial}[u]$ con los valores de $\text{especial}[x] + \text{Distancia}(x, u)$ para todo $x \in S$, incluyendo a v . Pero esa comparación ya se hizo cuando se añadió x a S por lo que $\text{especial}[x]$ no ha cambiado desde entonces. Por lo tanto, el cálculo del nuevo valor de $\text{especial}[u]$ se puede hacer comparando solamente su valor anterior con $\text{especial}[v] + \text{Distancia}(v, u)$. Como esto es lo que hace el algoritmo, el punto 2 también se cumple cuando se añade un nuevo nodo v a S .

Al finalizar el algoritmo todos los nodos excepto uno estarán en S y el camino más corto desde el origen hasta dicho nodo es un camino especial. Por lo que queda demostrado que el algoritmo de Dijkstra calcula los caminos de menor coste o longitud desde un nodo tomado como origen a los demás nodos del grafo.

Coste

En cuanto al coste, sea $G = \langle N, A \rangle$ un grafo de las características descritas en este apartado, con n nodos y a aristas. Las tareas de inicialización están en $O(n)$. La selección de v dentro del bucle **mientras** requerirá examinar $n - 1, n - 2, \dots, 2$ valores de $\text{especial}[]$ en los sucesivos pasos. Como el bucle **para** interno se ejecutará $n - 2, n - 3, \dots, 1$ veces, el tiempo requerido por el algoritmo de Dijkstra está en $O(n^2)$.

Este coste se puede reducir si se consigue evitar examinar todo el array $\text{especial}[]$ cada vez que se quiere asignar a v el nodo de coste mínimo. Para ello, se podría utilizar un montículo de mínimos que almacenara los nodos de los que aún no se ha determinado su camino de coste mínimo desde el origen. Es decir, el montículo contendría los nodos de C , y en la raíz estaría el que minimiza el valor de $\text{especial}[]$. La inicialización del montículo está en $O(n)$ (ver apartado del tema montículos). La instrucción que elimina v del conjunto C se traduce en eliminar la raíz del montículo, que está en $O(\log n)$. Si encontramos un nodo w tal que a través de v encontramos un camino menos costoso, debemos actualizar el montículo con w y ubicarlo según su valor de $\text{especial}[w]$, lo que requiere un tiempo que está en $O(\log n)$. Esto se hace como mucho una vez por cada arista del grafo. Por ello, se elimina la raíz del montículo $n - 2$ veces y se realizan operaciones flotar un máximo de a veces, lo que da un tiempo que está en $O((n + a) \log n)$. Si el grafo

es conexo, $n - 1 \leq a \leq n^2$. Si el grafo es disperso, el número de aristas es pequeño y cercano a n , y la implementación con montículo está en $O(a \log n)$. Si el grafo es denso, el coste pasa a estar en $O(n^2 \log n)$ y la implementación sin montículo es preferible.

3.3 Algoritmos voraces para planificación

En este apartado se van a describir dos problemas que están relacionados con planificar de forma óptima de una serie de tareas que deben llevarse a cabo por sólo un agente o servidor (persona o máquina). En el caso de la minimización del tiempo de espera en el sistema también se estudiará la generalización a más agentes o servidores.

3.3.1 Minimización del tiempo en el sistema

De forma general este problema se puede formular de la siguiente manera: hay una serie de n clientes o tareas que esperan un servicio de un único agente o servidor, y el tiempo que requerirá dar servicio a cada cliente es conocido, siendo t_i el tiempo requerido por el cliente i -ésimo y siendo $1 \leq i \leq n$. El objetivo es minimizar el tiempo medio de estancia de los clientes en el sistema. Como el valor de n es conocido, minimizar el tiempo medio de estancia en el sistema equivale a minimizar el tiempo total que están en el sistema todos los clientes.

Supongamos que tenemos tres clientes y sus tiempos de servicio son: $t_1 = 2$, $t_2 = 8$ y $t_3 = 4$. Las posibilidades de atender de forma consecutiva a los tres clientes son 6, y dan lugar a sus respectivos tiempos de estancia en el sistema:

| Orden de servicio | $T = \sum_{i=1}^n$ tiempo en el sistema del cliente i |
|-------------------|---|
| 1 2 3 | $2 + (2 + 8) + (2 + 8 + 4) = 26$ |
| 1 3 2 | $2 + (2 + 4) + (2 + 4 + 8) = \mathbf{22}$ |
| 2 1 3 | $8 + (8 + 2) + (8 + 2 + 4) = 32$ |
| 2 3 1 | $8 + (8 + 4) + (8 + 4 + 2) = 34$ |
| 3 1 2 | $4 + (4 + 2) + (4 + 2 + 8) = 24$ |
| 3 2 1 | $4 + (4 + 8) + (4 + 8 + 2) = 30$ |

Si nos fijamos en la primera secuencia, 1 2 3, el cliente 1 sólo está en el sistema su tiempo de servicio; el cliente 2 tiene que esperar a que atiendan al 1 y después permanece su tiempo de servicio; finalmente, el cliente 3 tiene que esperar a que atiendan al 1 y al 2 y después permanece su tiempo de servicio. Observando los resultados de T , se ve que varían según el orden de servicio, siendo el menor valor (en negrita) el que corresponde a haber atendido a los clientes en orden creciente de tiempos de servicio, y el mayor valor (34) el que corresponde a haber atendido a los clientes en orden decreciente de tiempos de servicio.

Para solucionar este problema se utiliza un algoritmo voraz que construye la secuencia ordenada óptima de clientes, y cuya función de selección elige en cada paso al cliente

que requiera el menor tiempo de servicio de entre los todavía no atendidos, y lo añade al final de la secuencia ordenada construida. Este algoritmo es muy sencillo y consistiría básicamente en ordenar los clientes por orden no decreciente de tiempos de servicio.

Demostración de optimalidad

Supongamos que hay n clientes numerados de 1 a n . Sea $S = s_1, s_2, \dots, s_n$ una permutación de los valores de 1 a n , y sea $e_i = t_{s_i}$ el tiempo de estancia o de servicio en el sistema del cliente i -ésimo. El tiempo total que están en el sistema todos los clientes es:

$$T(S) = e_1 + (e_1 + e_2) + (e_1 + e_2 + e_3) + \dots = ne_1 + (n-1)e_2 + (n-2)e_3 + \dots$$

$$= \sum_{k=1}^n (n-k+1)e_k$$

Supongamos que la permutación S no organiza a los clientes en orden creciente de tiempos de servicio. Entonces, habrá al menos dos enteros a y b tales que $a < b$ y $e_a > e_b$. Si se intercambia la posición de los clientes a y b se obtiene una nueva secuencia de servicio S' cuyo tiempo total en el sistema será:

$$= T(S') = (n-a+1)e_b + (n-b+1)e_a + \sum_{k=1, k \neq a, b}^n (n-k+1)e_k$$

Si restamos los tiempos de ambas permutaciones se obtiene:

$$\begin{aligned} T(S) - T(S') &= (n-a+1)(e_a - e_b) + (n-b+1)(e_b - e_a) \\ &= (n-a+1 - (n-b+1))(e_a - e_b) \\ &= (b-a)(e_a - e_b) > 0. \end{aligned}$$

Como sabemos que $a < b$ y que $e_a > e_b$ la permutación S' requiere menor tiempo que la permutación S . Es decir, cualquier permutación que no sea en orden creciente de tiempos de servicio se puede mejorar, de ahí que toda permutación cuyos clientes o tareas están ordenados por tiempos crecientes es óptima.

Queda el caso de cuando $a < b$ y $e_a = e_b$. En este caso el orden entre a y b es indiferente ya que se pueden intercambiar los valores e_a y e_b sin que se altere el valor de la suma total de tiempos.

Coste

Como básicamente lo que hace el algoritmo es ordenar los clientes por orden no decreciente de tiempo de servicio el coste está en $O(n \log n)$.

Generalización a a agentes o servidores

Este mismo problema se puede generalizar a un sistema con a agentes iguales. En este caso, además del orden de servicio que sería el mismo que para un sólo agente, hay que determinar cuántos clientes o tareas se asignan a cada agente o servidor. Si a un agente ag se le asignan p clientes o tareas y a otro agente ag' se le asignan $p+l$ clientes, siendo $l > 1$, el tiempo de servicio del primer cliente asignado a ag' se multiplica por un factor $p+l$, ya que su tiempo de servicio influye en el resto de tareas asignadas a ese agente, mientras que si se trasladara al primer puesto de los clientes de ag , desplazando a los otros p clientes, el factor multiplicativo sería $p+1$ y $p+1 < p+l$, por lo que el tiempo total decrecería. De ahí que una planificación óptima en un sistema con a agentes iguales debe mantener equilibrada la carga de todos los agentes. Así, la diferencia en el número de clientes asignados a cada agente debe ser a lo sumo de 1, por lo que la carga de cada agente será de $(n \text{ div } a)$ o de $(n \text{ div } a + 1)$. Si consideramos los clientes ordenados por orden no decreciente de tiempos ($t_1 \leq t_2 \leq \dots \leq t_n$), para una planificación S el reparto sería:

$$\begin{aligned} s_1 : & \quad t_1, t_{a+1}, t_{2a+1}, \dots \\ s_2 : & \quad t_2, t_{a+2}, t_{2a+2}, \dots \\ & \vdots \\ s_a : & \quad t_a, t_{2a}, t_{3a}, \dots \end{aligned}$$

Así, al agente s_j le correspondería la secuencia de clientes $t_1^j, t_2^j, \dots, t_{n_j}^j$, siendo $t_k^j = t_{(k-1)a+j}$, para $1 \leq k \leq n_j$ y

$$n_j = \begin{cases} n \text{ div } a + 1 & \text{si } j \leq (n \text{ mod } a) \\ n \text{ div } a & \text{si } j > (n \text{ mod } a) \end{cases}$$

El tiempo total en el sistema de S será:

$$T(S) = \sum_{j=1}^a \sum_{k=1}^{n_j} (n_j - k + 1) t_{(k-1)a+j}$$

3.3.2 Planificación con plazos

El problema consiste en que hay n tareas o trabajos; cada trabajo i en el caso de que se realice antes de su fecha tope f_i , siendo $f_i > 0$, permite obtener un beneficio b_i , siendo $b_i > 0$. En este tipo de problemas se da que:

- Para cualquier trabajo i el beneficio b_i se gana si y solo si el trabajo se realiza antes o coincidiendo con su fecha tope f_i .

- El trabajo se realiza en una máquina que consume una unidad de tiempo y solo hay una máquina disponible, es decir, en un instante de tiempo solo se puede ejecutar una tarea.

El objetivo es seleccionar los trabajos y la secuencia en la que se deben realizar para que el beneficio total sea máximo.

Una solución S o conjunto de tareas es *factible* si existe al menos una secuencia que permite que todas las tareas del conjunto se puedan completar antes de su fecha límite o plazo. El valor de una solución factible es la suma de los beneficios de los trabajos que contiene:

$$\sum_{i \in S} b_i$$

Una solución óptima es una solución factible con valor o beneficio máximo.

Veamos un ejemplo con $n = 4$ y los siguientes beneficios y plazos:

| i | 1 | 2 | 3 | 4 |
|-------|-----|----|----|----|
| f_i | 2 | 1 | 2 | 1 |
| b_i | 100 | 10 | 15 | 25 |

Las soluciones factibles, la secuencia de realización y los beneficios obtenidos serían:

| Solución factible | Secuencia | Beneficio |
|-------------------|-----------|------------|
| (1,2) | 2,1 | 110 |
| (1,3) | 1,3 o 3,1 | 115 |
| (1,4) | 4,1 | 125 |
| (2,3) | 2,3 | 25 |
| (3,4) | 4,3 | 40 |
| (1) | 1 | 100 |
| (2) | 2 | 10 |
| (3) | 3 | 15 |
| (4) | 4 | 25 |

La secuencia 1,2 no es factible ya que el trabajo 2 no se puede realizar en la unidad de tiempo 2, y lo mismo ocurre con la secuencia 1,4. Se observa que la secuencia óptima es la 4,1 con un beneficio de 125.

El algoritmo voraz para resolver este problema tiene la siguiente función de selección: considerar los trabajos en orden decreciente de beneficios siempre que el conjunto de trabajos sea una solución factible. En cada paso el siguiente trabajo que se selecciona será el de máximo beneficio, siempre que al unirlo al conjunto solución éste sea factible.

Lema 3.3.1 Si S es un conjunto de trabajos, entonces S es factible si y solo si la secuencia obtenida ordenando los trabajos en orden no decreciente de fechas tope es factible.

Para demostrar este lema supongamos que S es factible, entonces existe al menos una secuencia factible de los trabajos de S . Supongamos que en esa secuencia factible el trabajo u se planifica antes que el trabajo v siendo $f_v < f_u$. Si intercambiamos los dos trabajos, el trabajo v se adelanta comenzando antes de su fecha tope. Al retrasar u a la posición de v y al ser $f_v < f_u$ y la secuencia factible, la nueva secuencia sigue siendo factible. Supongamos ahora que la secuencia $1, 2, \dots, k$ de S no es factible. Esto significa que al menos un trabajo está planificado después de su plazo. Sea l una cualquiera de esas tareas tal que $f_l \leq l - 1$. Como las tareas están ordenadas por orden no decreciente de plazos, al menos l tareas tienen como fecha final $l - 1$ o una fecha anterior. Por lo que en cualquier caso la última siempre está planificada más tarde que su fecha tope.

Así, basta comprobar una secuencia en orden no decreciente de fechas para saber si un conjunto de tareas es o no factible.

Una descripción de alto nivel del algoritmo voraz que resuelve este problema es la siguiente:

```
fun PlanificacionPlazos (F[1..n], n): Vector[1..n] de natural
    S ← {1}
    para i = 2 hasta n hacer
        si los trabajos en S ∪ {i} constituyen una secuencia factible entonces
            S ← S ∪ {i}
        fsi
    fpara
    dev S
ffun
```

El array $F[]$ almacena las fechas tope de realización de los n trabajos ordenados en orden decreciente de beneficios.

Demostración de optimalidad

Se trata de demostrar que el algoritmo voraz que considera los trabajos en orden decreciente de beneficios, siempre que el conjunto de trabajos sea una solución factible, encuentra una planificación óptima.

Supongamos que se quieren ejecutar un conjunto I y un conjunto J de trabajos o tareas. El conjunto I consta de las tareas $\{g, h, i, k, l\}$ y el conjunto J consta de las tareas $\{l, m, o, g, p, q, i\}$. Supongamos también que el conjunto J es óptimo. Sean S_I y S_J secuencias factibles de los respectivos conjuntos, que podrían incluir huecos en su planificación:

| | | | | | | | |
|---------|-----|-----|-----|-----|-----|-----|-----|
| $S_I :$ | g | h | i | k | l | | |
| $S_J :$ | l | m | o | g | p | q | i |

Se pueden reorganizar los trabajos de S_I y S_J de manera que se obtienen las secuencias S''_I y S''_J que también son factibles, en las que los trabajos comunes a ambas secuencias se planifican en la misma unidad de tiempo. También en estas secuencias podría haber huecos:

| | | | | | | | |
|-----------|-----|-----|-----|-----|-----|-----|-----|
| $S''_I :$ | k | h | | g | l | | i |
| $S''_J :$ | p | m | o | g | l | q | i |

Supongamos que un trabajo a aparece en las 2 secuencias factibles S_I y S_J , planificado respectivamente en los tiempos t_I y t_J . En el caso de que $t_I = t_J$ no hay cambios posibles. Supongamos que $t_I < t_J$. Como la secuencia S_J es factible, el plazo para la tarea a no es anterior a t_J y se atrasa la tarea a del tiempo t_I al tiempo t_J en su secuencia S_I . Se pueden dar dos situaciones:

- Si hay un hueco en la secuencia S_I en la unidad de tiempo t_I se pasa la tarea a a dicho hueco.
- Si ya hay una tarea b planificada en ese tiempo t_I en S_I , se modifica S_I intercambiando las tareas a y b en la secuencia.

La secuencia resultante sigue siendo factible ya que a se sigue ejecutando antes de su plazo y en el caso de que exista b se adelanta. Ahora a está planificada en ambas secuencias en la unidad de tiempo t_J . En el caso de que $t_I > t_J$ se realiza lo mismo pero en este caso con la secuencia S_J . Una vez realizado este procedimiento con la tarea a , ésta se queda ya en dicha unidad de tiempo.

Si las secuencias S_I y S_J tienen r tareas en común, tras un máximo de r modificaciones de S_I o S_J , las tareas comunes a ambas secuencias estarán planificadas al mismo tiempo en S''_I y S''_J . Si $I \neq J$ las secuencias resultantes S''_I y S''_J pueden no ser iguales. Supongamos que en una determinada unidad de tiempo la tarea planificada en S''_I es distinta de la planificada en S''_J :

- Si hay una tarea a en S''_I y en la misma unidad de tiempo en S''_J hay un hueco, entonces a no pertenece a J . El conjunto $J \cup \{a\}$ sería factible, ya que se podría poner a en el hueco y el beneficio sería mayor que en J . Pero por hipótesis partimos de que J es óptimo, por lo que no es posible.
- Si hay una tarea b en S''_J y en la misma unidad de tiempo en S''_I hay un hueco, el conjunto $J \cup \{b\}$ sería factible, por lo que el algoritmo habría incluido b en I . Como el algoritmo no lo hizo entonces tampoco es posible.
- El tercer caso es que una tarea a esté planificada en S''_I y en la misma unidad de tiempo en S''_J hay una tarea distinta b . En este caso a no pertenece a J y b no pertenece a I . Las posibilidades son:

- Si $b_a > b_b$ se podría sustituir a por b en J y mejorar el beneficio. Pero esto no es posible ya que J es óptima.
- Si $b_a < b_b$ el algoritmo voraz habría seleccionado a antes de considerar a b puesto que $(I \setminus \{a\}) \cup \{b\}$ sería factible. Esto no es posible porque el algoritmo voraz no incluyó a b en I .
- Si $b_a = b_b$ aportan la misma ganancia en las secuencias.

Por lo que para toda unidad de tiempo de las secuencias S''_I y S''_J o no planifican tareas, o planifican la misma, o bien planifican dos tareas distintas con igual beneficio. El beneficio total de I es por lo tanto igual al beneficio del conjunto óptimo J , así que I también es óptimo.

Algoritmo detallado

Veamos a continuación un algoritmo más detallado. Supongamos que las tareas están numeradas de forma que $b_1 \geq b_2 \geq \dots \geq b_n$. El array $f[1..n]$ almacena las fechas tope de los trabajos en orden decreciente de beneficios. El array $S[1..k]$ almacena los elementos de S , siendo $S[i]$ el i -ésimo trabajo en la solución óptima. Se debe cumplir que: $f[S[1]] \leq f[S[2]] \leq \dots \leq f[S[k]]$. Además, vamos a suponer que $n > 0$ y que $f_i > 0$ para $1 \leq i \leq n$, lo que nos permite añadir al array f un elemento centinela $f[0] \leq 0$ para simplificar las comprobaciones del algoritmo.

tipo VectorNat = matriz[0..n] de natural

fun PlanificacionPlazosDetallado (f: VectorNat, n: natural): VectorNat, natural

var

 S: VectorNat

fvar

 S[0] $\leftarrow 0$ {elemento centinela para facilitar la inserción}

 S[1] $\leftarrow 1$ {se incluye el trabajo 1 que es el de máximo beneficio}

 k $\leftarrow 1$ {k: nº de elementos en S}

para i = 2 **hasta** n **hacer**

 r $\leftarrow k$ {se busca una posición válida para i}

mientras (f[S[r]] > f[i]) \wedge (f[S[r]] \neq r) **hacer**

 r $\leftarrow r - 1$

fmientras

si (f[S[r]] \leq f[i]) \wedge (f[i] > r) **entonces**

para q = k **hasta** r + 1 **incr** = -1 **hacer**

 S[q+1] \leftarrow S[q]

fpara

 S[r+1] \leftarrow i

 k $\leftarrow k + 1$

fsi

fpara
dev S, k
ffun

Para cada tarea i -ésima, el algoritmo comprueba que se pueda insertar sin desplazar a las tareas ya planificadas más allá de su plazo. Si esto es posible i se inserta, sino se descarta.

Veamos la traza del algoritmo con los siguientes datos: $n = 5$, $b_i = (20, 15, 10, 5, 1)$ y $f_i = (2, 2, 1, 3, 3)$

| Paso | k | i | S |
|----------------|---|---|-----------------|
| Inicialización | 1 | | 1 _ _ _ _ |
| 1 | | 2 | |
| | | 2 | r = 1 1 2 _ _ _ |
| 2 | | 3 | |
| | | | r = 2 1 2 _ _ _ |
| 3 | | 4 | |
| | | 3 | r = 2 1 2 4 _ _ |
| 4 | | 5 | |
| | | | r = 3 |

La planificación óptima es la secuencia de tareas 1,2,4 que tendría un beneficio de 40.

Coste

La ordenación de $f[]$ en orden decreciente de beneficios requiere un coste $\Theta(n \log n)$, siendo n el número de tareas. El caso peor del algoritmo es cuando coincide que todas las tareas en f también están ordenadas por orden decreciente de plazos y todas ellas se pueden incluir en la planificación. Esto supone que para cada tarea i el algoritmo evalúa las $i - 1$ tareas ya seleccionadas, hasta que encuentra su ubicación y realiza los desplazamientos oportunos. Por lo tanto este algoritmo está en $O(n^2)$.

Algoritmo mejorado

Hay un algoritmo voraz más eficiente para resolver este problema que utiliza un método diferente de determinar la factibilidad de una solución parcial. Un conjunto S de n tareas es factible si y solo si se puede construir una secuencia factible que incluya a todas las tareas de S de la siguiente manera: empezando con una planificación vacía, para cada tarea $i \in S$ se planifica i en el instante t , siendo t el mayor entero tal que $1 \leq t \leq \min(n, f_i)$ y la tarea que se ejecutará en t no se ha decidido todavía.

Esto supone que cada tarea se añade a la planificación en el instante de tiempo más tardío posible, sin que se pase su plazo. Si no se encuentra un instante de tiempo de

estas características la tarea en cuestión no se añade a S , ya que el conjunto S resultante no sería factible.

Veamos la demostración de que este método de determinar la factibilidad de una solución parcial conduce a una solución óptima. El que cada tarea i se añada a la planificación en el instante de tiempo más tardío posible, se puede formalizar indicando que el día elegido será:

$$t(i) = \max\{t \mid 1 \leq t \leq \min\{n, f_i\} \wedge (\forall j : 1 \leq j < i : t \neq t(j))\}.$$

Cuando se intenta añadir una nueva tarea, la secuencia que se está construyendo contiene al menos un hueco. Supongamos que no se puede añadir una tarea i cuyo plazo sea f_i . Esto sólo puede ocurrir si todas las posiciones desde la 1 hasta la $\min\{n, f_i\}$ están ya ocupadas. Sea $s > \min\{n, f_i\}$ el mayor entero tal que la posición $t = s$ está vacía. La planificación incluye por lo tanto $s - 1$ tareas, ninguna tarea con plazo s y quizás otras con plazos posteriores a s . La tarea que se quiere añadir también tiene un plazo anterior a s . Como la planificación contiene al menos s tareas cuyos plazos son $s - 1$ o anteriores, la última tarea llegará tarde, lo que demuestra el “solo si” de la hipótesis.

Para implementar esta prueba de factibilidad definimos $libre(i) = \max\{t \leq i \mid t \text{ libre}\}$, es decir, es el primer predecesor libre de i . De esta manera se definen conjuntos de posiciones, de forma que las posiciones i y j están en el mismo conjunto si y solo si $libre(i) = libre(j)$. Se define además una posición ficticia 0 que siempre estará libre. Así, la tarea i debería realizarse en el día $libre(f_i)$ ya que representa el último día libre que respeta su plazo.

Los pasos principales del algoritmo mejorado son:

- Inicialmente cada posición o instante de tiempo $0, 1, 2, 3, \dots, p$ está en un conjunto diferente y $libre(\{i\}) = i, 0 \leq i \leq p$
- Si se quiere añadir una tarea con plazo f se busca el conjunto que contenga a f . Sea K dicho conjunto. Si $libre(K) = 0$ se rechaza la tarea; en caso contrario se realizan las siguientes acciones:
 - Se asigna la tarea al instante de tiempo $libre(K)$.
 - Se busca el conjunto que contenga $libre(K) - 1$. Sea L dicho conjunto.
 - Se fusionan K y L . El valor de $libre()$ para este nuevo conjunto es el valor que tenía $libre(L)$.

Asumiendo que las tareas están ordenadas en orden decreciente de beneficios la descripción del algoritmo es la siguiente:

tipo VectorNat = matriz[0..n] de natural

fun PlanificacionPlazosMejorado (f: VectorNat, n: natural): VectorNat, natural

var

```

S, libre: VectorNat
fvar
p  $\leftarrow \min(n, \max\{f[i] \mid 1 \leq i \leq n\})$ 
para i = 0 hasta n hacer
    S[i]  $\leftarrow 0$ 
    libre[i]  $\leftarrow i$ 
    Iniciar conjunto i
fpara
para i = 1 hasta n hacer
    k  $\leftarrow \text{buscar}(\min(p, f[i]))$ 
    pos  $\leftarrow \text{libre}(k)$ 
    si pos  $\neq 0$  entonces
        S[pos]  $\leftarrow i$ 
        l  $\leftarrow \text{buscar}(pos - 1)$ 
        libre[k]  $\leftarrow \text{libre}[l]$ 
        fusionar(k, l) {asignar la etiqueta k o l}
    fsi
fpara
k  $\leftarrow 0$ 
para i = 1 hasta n hacer
    si S[i] > 0 entonces
        k  $\leftarrow k + 1$ 
        S[k]  $\leftarrow S[i]$ 
    fsi
fpara
dev S, k
ffun

```

Veamos la traza del algoritmo con los siguientes datos: $n = 5$, $b_i = (20, 15, 10, 5, 1)$ y $f_i = (2, 2, 1, 3, 3)$

| Paso | p | k | pos | l | libre[] | S |
|----------------|---|---|-----|---|-------------------------|-----------|
| Inicialización | 3 | | | | {0} {1} {2} {3} {4} {5} | 0 0 0 0 0 |
| 1 | | 2 | 2 | 1 | {0} {1, 2} {3} {4} {5} | 0 1 0 0 0 |
| 2 | | 2 | 1 | 0 | {0, 1, 2} {3} {4} {5} | 2 1 0 0 0 |
| 3 | | 1 | 0 | | {0, 1, 2} {3} {4} {5} | 2 1 0 0 0 |
| 4 | | 3 | 3 | 2 | {0, 1, 2, 3} {4} {5} | 2 1 4 0 0 |
| 5 | | 3 | 0 | | {0, 1, 2, 3} {4} {5} | 2 1 4 0 0 |
| Comprimir S[] | | | | | {0, 1, 2, 3} {4} {5} | 2 1 4 0 0 |

Coste del algoritmo mejorado

La ordenación de las tareas por beneficios decrecientes está en $O(n \log n)$. La fase de inicialización está en $O(n)$. Después, se realizan como máximo $2n$ operaciones *buscar()* y n operaciones *fucionar()*, lo que es esencialmente lineal en n . En total nos queda que el algoritmo está en $O(n \log n)$.

3.4 Almacenamiento óptimo en un soporte secuencial

Se dispone de n programas y hay que almacenarlos en un soporte secuencial de longitud L . Cada programa p_i tiene una longitud l_i , siendo $1 \leq i \leq n$. Se podrán almacenar los n programas en el soporte si y solo si la suma de las longitudes de todos los programas es como mucho L . Asumiremos que para poder acceder a un programa p_i , independientemente de donde esté ubicado en la cinta, habrá que posicionar el mecanismo de acceso al comienzo del soporte. De ahí, que si un programa ocupa la posición x_i , el tiempo de acceso a ese programa será la suma del tiempo que tarda en avanzar el mecanismo de acceso hasta x_i , más el tiempo de lectura de p_i que es l_i . Si todos los programas se recuperan o acceden con similar frecuencia, el tiempo medio de acceso es:

$$T = \frac{\sum_{i=1}^n (x_i + l_i)}{n}$$

El objetivo es encontrar una secuencia de almacenamiento de los programas que minimice el tiempo medio de acceso.

Si los programas están almacenados en el orden $I = i_1, i_2, \dots, i_n$, el tiempo t_j necesario para recuperar el programa i_j es proporcional a $\sum_{1 \leq k \leq j} l_{i_k}$, por lo que minimizar el tiempo medio de acceso es equivalente a minimizar:

$$D(I) = \sum_{1 \leq j \leq n} \sum_{1 \leq k \leq j} l_{i_k}$$

Dada una secuencia $D(I)$ la estrategia voraz deberá elegir el siguiente programa que minimice el crecimiento en $D(I)$. Si ya se tiene la permutación i_1, i_2, \dots, i_r al añadir el programa j a la permutación se tiene $i_1, i_2, \dots, i_r, i_{r+1} = j$. Esto incrementa el valor de $D(I)$ en $\sum_{1 \leq k \leq r} l_{i_k} + l_j$. Como $\sum_{1 \leq k \leq r} l_{i_k}$ es fijo e independiente de j , se observa que se minimiza el incremento de D si el siguiente programa que se selecciona es el de menor longitud de los restantes.

Por lo tanto la estrategia voraz consistirá en seleccionar los programas en orden no decreciente de longitudes. Resulta sencillo identificar los elementos del esquema voraz para este problema concreto:

- Resolución de un problema de forma óptima: almacenar una serie de programas en un soporte secuencial de manera que el tiempo medio de acceso sea mínimo.
- *Conjunto inicial de candidatos*: los n programas.

- *Conjunto de candidatos seleccionados*: inicialmente el conjunto de candidatos ya escogidos está vacío.
- *Solución*: si la suma de las longitudes de todos los programas es como mucho L se alcanzará la solución cuando los n programas se hayan seleccionado en el orden adecuado y se hayan almacenado.
- *Factible*: si la suma de las longitudes de todos los programas es como mucho L el conjunto resultante siempre es factible.
- La *función de selección* se traduce en la selección de los programas en orden no decreciente de longitudes.
- La *función objetivo* no aparece explícitamente en el algoritmo.

El algoritmo voraz quedaría:

```
fun AlmacenaProgramasSoporteSec (c: conjuntoCandidatos): conjuntoCandidatos
    Ordenar c en orden no decreciente de longitudes
    sol ← ∅
    mientras c ≠ ∅ hacer
        x ← seleccionar(c)
        c ← c - {x}
        sol ← sol ∪ {x}
    fmientras
    devolver sol
ffun
```

La función *seleccionar()* selecciona los programas en el orden establecido.

Demostración de optimalidad

Se trata de demostrar que esta estrategia voraz conduce a una solución óptima. Es decir, demostrar que si $l_1 \leq l_2 \leq \dots \leq l_n$ entonces la ordenación $i_j = j$ siendo $1 \leq j \leq n$ minimiza $\sum_{k=1}^n \sum_{j=1}^k l_{i_j}$ sobre todas las posibles permutaciones de i_j .

Sea $I = i_1, i_2, \dots, i_n$ cualquier permutación del conjunto de índices $\{1, 2, 3, \dots, n\}$. Entonces:

$$D(I) = \sum_{k=1}^n \sum_{j=1}^k l_{i_j} = \sum_{1 \leq k \leq n} (n-k+1)l_{i_k}$$

Si existen a y b tales que $a < b$ y $l_{i_a} > l_{i_b}$ entonces, intercambiando i_a e i_b obtenemos una permutación I' en la que:

$$D(I') = \sum_{k \neq a \wedge k \neq b}^n (n-k+1)l_{i_k} + (n-a+1)l_{i_b} + (n-b+1)l_{i_a}$$

Si restamos $D(I')$ de $D(I)$ tenemos:

$$D(I) - D(I') = (n - a + 1)(l_{i_a} - l_{i_b}) + (n - b + 1)(l_{i_b} - l_{i_a}) = (b - a)(l_{i_a} - l_{i_b}) > 0$$

Por lo que si colocamos primero el programa más corto el sumatorio reduce su valor. De ahí que si los programas están ordenados en longitudes no decrecientes el sumatorio no puede ser reducido y por lo tanto minimiza el valor de D .

Coste

La implementación del conjunto de programas se puede realizar con un array. El coste de ordenar el array de programas es $O(n \log n)$. La implementación del conjunto solución también se puede realizar con un array de n elementos, en el que en la posición i se almacena el identificador del programa que se graba en la cinta en la posición i -ésima. Así, el coste del bucle mientras está en $O(n)$, por lo que el coste global del algoritmo está en $O(n \log n)$.

3.4.1 Generalización a n soportes secuenciales

Si en lugar de un soporte secuencial se tienen m soportes, S_0, S_1, \dots, S_{m-1} siendo $m > 1$, y un único mecanismo de acceso, habrá que distribuir los programas en todos ellos. El objetivo sigue siendo el mismo, encontrar las secuencias de almacenamiento de los programas que minimicen el tiempo medio de acceso. El resultado será la permutación u orden de almacenamiento para cada soporte. Sea I_j la permutación u orden de programas para el soporte j , que tendrá su correspondiente $D(I_j)$. El tiempo total de acceso será:

$$TD = \sum_{j=0}^{m-1} D(I_j)$$

El objetivo será almacenar los programas de tal manera que se minimice el valor TD . La función de selección de los programas es igual que en el caso de un único soporte, en orden no decreciente de longitudes. Una vez seleccionado el programa a almacenar, se grabará en el soporte que minimice el incremento de TD . Este soporte será el que tenga menos parte ocupada hasta el momento. Si hay más de uno en estas condiciones, se seleccionará el de número de índice más bajo. Como los programas están ordenados de manera que $l_1 \leq l_2 \leq \dots \leq l_n$, entonces los primeros m programas se almacenarán respectivamente en los soportes S_0, S_1, \dots, S_{m-1} . Los siguientes m programas también se almacenarán respectivamente en los soportes S_0, S_1, \dots, S_{m-1} . En general, el programa i se almacenará en el soporte $S_i \bmod m$. De esta manera, en cada soporte los programas están almacenados en orden no decreciente de sus longitudes.

3.5 Problema de la mochila con objetos fraccionables

Se dispone de n objetos y una mochila. El objeto i tiene un peso positivo p_i , un valor positivo v_i y la mochila solo puede almacenar un peso máximo M . Como los objetos se pueden fraccionar, una fracción x_i del objeto i , siendo $0 \leq x_i \leq 1$ añadirá a la mochila un peso de $x_i p_i$ y un valor de $x_i v_i$. El objetivo es llenar la mochila de manera que se maximice el valor total de los objetos almacenados, teniendo en cuenta la limitación del peso máximo de la mochila. Formalmente, el problema se puede enunciar de la siguiente manera:

$$\text{maximizar } \sum_{i=1}^n x_i v_i \text{ con la restricción } \sum_{i=1}^n x_i p_i \leq M, \text{ y } 0 \leq x_i \leq 1, 1 \leq i \leq n$$

En el esquema general voraz, los candidatos serían los objetos, una solución factible es cualquier conjunto $\{x_1, x_2, \dots, x_n\}$ que satisfaga las restricciones indicadas anteriormente, y la función objetivo es el valor total de los objetos almacenados en la mochila. Veamos ahora cuál podría ser la función de selección. Como los objetos son fraccionables en cualquier fracción, una solución óptima debe llenar exactamente la mochila, porque en caso contrario, y siendo $\sum_{i=1}^n p_i > M$, siempre se podría añadir una fracción de alguno de los objetos restantes, incrementando el valor total de la mochila. Una vez que se determine en qué orden deben seleccionarse los objetos, se pondrá la máxima fracción posible del objeto seleccionado en la mochila y se continuará el proceso hasta que la mochila esté llena.

Consideremos la siguiente instancia del problema de la mochila: $n = 3$, $M = 20$, los pesos y valores de los objetos son $(p_1, p_2, p_3) = (18, 15, 10)$ y $(v_1, v_2, v_3) = (25, 24, 15)$, y se dispone de tres funciones de selección posibles: (1) seleccionar el objeto más valioso de los restantes, (2) el de menos peso, y (3) el objeto cuyo valor por unidad de peso sea el mayor posible.

| Función selección | x_i | $\sum_{i=1}^n x_i v_i$ | $\sum_{i=1}^n x_i p_i$ |
|---------------------|-----------|----------------------------|------------------------|
| maximizar v_i | 1 2/15 0 | $25 + 3.2 = 28.2$ | $18 + 2 = 20$ |
| minimizar p_i | 0 10/15 1 | $16 + 15 = 31$ | $10 + 10 = 20$ |
| maximizar v_i/p_i | 0 1 5/10 | $24 + 7.5 = \mathbf{31.5}$ | $15 + 5 = 20$ |

Si la función de selección consiste en seleccionar el objeto más valioso de los restantes, primero se seleccionaría el objeto x_1 entero con un valor de 25, después, como el siguiente más valioso, el x_2 , no cabe entero se tomaría una fracción calculando el peso que queda disponible en la mochila ($20 - 18$) y dividiéndolo por el peso del objeto que es 15. Esto nos da un valor de 0.133 que es la fracción del objeto que se toma para completar el peso y calcular el valor que aporta el objeto. Si la función selecciona el objeto de menor peso de los restantes, primero se seleccionaría el objeto x_3 entero con un valor de 15, después, como el siguiente con menos peso, el x_2 , no cabe entero se tomaría una fracción calculando el peso que queda disponible en la mochila ($20 - 15$)

y dividiéndolo por el peso del objeto que es 15. Esto nos da un valor de 0.33 que es la fracción del objeto que se toma para completar el peso y calcular el valor que aporta el objeto. Finalmente, si la función selecciona el objeto cuyo valor por unidad de peso sea el mayor posible de los restantes, nos quedarían unos valores (1.388 1.6 1.5) para los tres objetos. Primero se seleccionaría el objeto x_2 entero con un valor de 24, después, como el siguiente con menos peso, el x_3 , no cabe entero se tomaría una fracción calculando el peso que queda disponible en la mochila (20 – 15) y dividiéndolo por el peso del objeto que es 10. Esto nos da un valor de 0.5 que es la fracción del objeto que se toma para completar el peso y calcular el valor que aporta el objeto.

De las tres estrategias, la que obtiene el máximo beneficio total (31.5) es la que considera los objetos en orden no creciente del cociente del valor por peso, y ésta es precisamente la función de selección del algoritmo voraz que resuelve de forma óptima este problema: considerar en cada paso los objetos que tienen el máximo valor por unidad de peso. A continuación se muestra el algoritmo:

```

tipo VectorNat = matriz[0..n] de natural
tipo VectorRea = matriz[0..n] de real
fun MochilaObjetosFraccionables (p: VectorNat, v: VectorNat, M: natural): VectorRea
    var
        x: VectorRea
        peso: natural
    fvar
        Ordenar objetos en orden no decreciente de  $v_i/p_i$ 
        peso  $\leftarrow$  0
    para i = 1 hasta n hacer
        x[i]  $\leftarrow$  0
    fpara
    mientras peso < M hacer
        i  $\leftarrow$  mejor objeto de los restantes
        si peso + p[i]  $\leq$  W entonces
            x[i]  $\leftarrow$  1
            peso  $\leftarrow$  peso + p[i]
        sino
            x[i]  $\leftarrow$  (M - peso)/ p[i]
            peso  $\leftarrow$  M
        fsi
    fmientras
    dev x
ffun
```

Demostración de optimalidad

Se trata de comprobar que efectivamente la función de selección propuesta conduce a que el algoritmo calcule una solución óptima.

Teorema 3.5.1 *Si los objetos se seleccionan en orden decreciente de v_i/p_i es decir $v_1/p_1 \geq v_2/p_2 \geq \dots \geq v_n/p_n$, entonces el algoritmo voraz anterior encuentra una solución óptima.*

Supongamos que los objetos están ordenados en orden decreciente de v_i/p_i . Sea $X = (x_1, x_2, \dots, x_n)$ la solución calculada por el algoritmo voraz. Si todos los x_i son iguales a 1, entonces la solución es óptima. En caso contrario, sea j el menor índice tal que $x_j < 1$. Del funcionamiento del algoritmo se sigue que $x_i = 1$ cuando $1 \leq i < j$, que $x_i = 0$ cuando $j < i \leq n$, que $0 \leq x_j < 1$, y que $\sum_{i=1}^n x_i p_i = M$. Sea $V(X) = \sum_{i=1}^n x_i v_i$ el valor de la solución X .

Sea $Y = (y_1, y_2, \dots, y_n)$ una solución factible cualquiera. Se trata de comparar X con Y para demostrar que $V(X) \geq V(Y)$ de lo que se deduce que X es óptima.

Si Y es factible, entonces $\sum_{i=1}^n y_i p_i \leq M$ y por lo tanto $\sum_{i=1}^n (x_i - y_i)v_i \geq 0$. Sea $V(Y) = \sum_{i=1}^n y_i v_i$ el valor de la solución Y ,

$$V(X) - V(Y) = \sum_{i=1}^n (x_i - y_i)v_i = \sum_{i=1}^n (x_i - y_i)p_i \frac{v_i}{p_i}$$

tenemos tres posibilidades a considerar:

- Cuando $i < j$ entonces $x_i = 1$ y por lo tanto $x_i - y_i \geq 0$, mientras que $v_i/p_i \geq v_j/p_j$ debido a la ordenación de los objetos.
- Cuando $i > j$ entonces $x_i = 0$ y por lo tanto $x_i - y_i \leq 0$, mientras que $v_i/p_i \leq v_j/p_j$.
- Si $i = j$ entonces $v_i/p_i = v_j/p_j$.

Por lo que en todos los casos se tiene que $(x_i - y_i)(v_i/p_i) \geq (x_i - y_i)(v_j/p_j)$ por lo que:

$$V(X) - V(Y) \geq \sum_{i=1}^n (x_i - y_i)p_i \frac{v_j}{p_j} = \frac{v_j}{p_j} \sum_{i=1}^n (x_i - y_i)p_i$$

Como $\frac{v_j}{p_j} > 0$ porque todos los valores y pesos son positivos y además se cumple que:

$$\sum_{i=1}^n x_i p_i = M \text{ y } \sum_{i=1}^n y_i p_i \leq M$$

se tiene:

$$V(X) - V(Y) \geq \sum_{i=1}^n (x_i - y_i)p_i = \sum_{i=1}^n x_i p_i - \sum_{i=1}^n y_i p_i \geq 0$$

Queda demostrado que ninguna solución factible puede tener un valor mayor que $V(X)$, por lo que X es una solución óptima.

Coste

El coste de ordenar los objetos en orden decreciente de v_i/p_i está en $O(n \log n)$. El orden del resto del algoritmo está en $O(n)$ por lo que el algoritmo completo está en $O(n \log n)$.

3.6 Mantenimiento de la conectividad

Una operadora de telecomunicaciones dispone de 8 nodos conectados todos entre sí por una tupida red de conexiones punto a punto de fibra óptica. Cada conexión $c(i, j)$ entre los nodos i y j , siendo $i, j \in \{1 \dots 8\}$ tiene un coste que viene dado por $c(i, j) = (i \times j) \bmod 6$. La operadora está en una situación económica complicada y necesita reducir gastos. Por ello, se plantea reducir las conexiones de forma que se mantenga la conectividad de la red a un coste mínimo.

Se trata de un problema de optimización en el que se quiere mantener la conectividad de la red, es decir que el grafo sea conexo, a un coste mínimo. Se trata por lo tanto de asociar al grafo (la red) un árbol de recubrimiento de coste mínimo. Para su resolución se puede usar cualquiera de los dos algoritmos vistos: Prim o Kruskal (ver los apartados 3.2.1 y 3.2.2 para revisar los algoritmos de Prim y Kruskal respectivamente).

Al indicar el enunciado que se trata de una red tupida en la que todos los nodos están conectados entre sí, se trata de un grafo denso, por lo que es adecuado representarlo mediante una matriz de adyacencia.

La red de conexiones se representaría mediante la siguiente matriz de adyacencia, que al corresponder a un grafo no dirigido es simétrica. Nótese que se representa solamente su triangular superior:

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 1 | - | 2 | 3 | 4 | 5 | 0 | 1 | 2 |
| 2 | | - | 0 | 2 | 4 | 0 | 2 | 4 |
| 3 | | | - | 0 | 3 | 0 | 3 | 0 |
| 4 | | | | - | 2 | 0 | 4 | 2 |
| 5 | | | | | - | 0 | 5 | 4 |
| 6 | | | | | | - | 0 | 0 |
| 7 | | | | | | | - | 2 |
| 8 | | | | | | | | - |

Para calcular el árbol de recubrimiento mínimo vamos a utilizar el algoritmo de Prim, en particular la descripción de alto nivel del algoritmo que nos va a facilitar aplicarlo a los datos del enunciado. Recuérdese que se utilizan dos conjuntos: uno con los nodos seleccionados NA , y el conjunto de aristas del árbol de recubrimiento mínimo AR . Se toma como raíz del árbol el nodo 1.

```

fun Prim ( $G = \langle N, A \rangle$ : grafo): conjunto de aristas
    AR  $\leftarrow \emptyset$ 
    NA  $\leftarrow \{\text{un nodo cualquiera de } N\}$ 
    mientras NA  $\neq N$  hacer
        Buscar  $\{u, v\}$  de coste mínimo tal que  $u \in \text{NA}$  y  $v \in N \setminus \text{NA}$ 
        AR  $\leftarrow \text{AR} \cup \{(u, v)\}$ 
        NA  $\leftarrow \text{NA} \cup \{v\}$ 
    fmientras
    dev AR
ffun

```

Veamos ahora la traza del algoritmo con la anterior matriz de adyacencia:

| Paso | arista seleccionada | NA | AR |
|--------|---------------------|-------------------|-------------------|
| Inicio | | {1} | \emptyset |
| 1 | {1,6} | {1,6} | AR $\cup \{1,6\}$ |
| 2 | {6,2} | {1,2,6} | AR $\cup \{6,2\}$ |
| 3 | {2,3} | {1,2,3,6} | AR $\cup \{2,3\}$ |
| 4 | {3,8} | {1,2,3,6,8} | AR $\cup \{3,8\}$ |
| 5 | {4,6} | {1,2,3,4,6,8} | AR $\cup \{4,6\}$ |
| 6 | {6,5} | {1,2,3,4,5,6,8} | AR $\cup \{6,5\}$ |
| 7 | {6,7} | {1,2,3,4,5,6,7,8} | AR $\cup \{6,7\}$ |

El coste total del árbol calculado es 0, ya que todas las aristas seleccionadas tienen coste 0. En cuanto al coste computacional, se remite al lector al apartado 3.2.1 para una explicación detallada.

3.7 Problema de mensajería urgente

Una empresa de mensajería urgente realiza buena parte de su transporte por carretera. Dispone de un servicio exprés, en el que se compromete con el cliente a llevar el paquete a su destino sin hacer ninguna otra entrega en el camino. La empresa conoce el número de kilómetros que puede hacer sin repostar cada vehículo de su parque móvil. También dispone de un mapa de carreteras que le permite calcular las distancias que hay entre las gasolineras en todas las posibles rutas. Así, dado un punto de origen y otro de destino, el transportista puede parar a repostar el menor número de veces posible. Se trata de diseñar un algoritmo voraz que determine en qué gasolineras debe parar el transportista para lograr el objetivo de repostar el menor número de veces posible.

Sea n el número de kilómetros que puede realizar sin repostar el vehículo que se va a utilizar para realizar el trasporte exprés. Suponemos que en la ruta entre el punto de recogida del paquete y el punto de destino existen G gasolineras numeradas del 0, en el punto de recogida, a $G - 1$ en el punto destino. Suponemos también que se dispone de un

vector con las distancias entre las gasolineras, $DG[1..G - 1]$, siendo $DG[i]$ el número de kilómetros que hay entre la gasolinera $i - 1$ e i . Nótese que el problema tendrá solución si ningún elemento del vector tiene un valor mayor que n , ya que en caso contrario el vehículo no podría llegar de una gasolinera a otra que distara más de n kilómetros. Hay que tener en cuenta que hay una única ruta posible.

Identifiquemos los elementos del esquema voraz para este problema concreto:

- Resolución de un problema de forma óptima: repostar el menor número de veces posible en la realización de un trayecto.
- *Conjunto inicial de candidatos*: las G gasolineras.
- *Conjunto de candidatos seleccionados*: inicialmente el conjunto de candidatos ya escogidos está vacío.
- *Solución*: se alcanzará la solución cuando se llegue al destino repostando según indique la función de selección.
- *Factible*: si la distancia entre las gasolineras de la solución es menor o igual que n el conjunto resultante siempre es factible.
- La *función de selección* se traduce en recorrer el mayor número de kilómetros sin repostar.
- La *función objetivo* no aparece explícitamente en el algoritmo.

Por tanto, la estrategia voraz que se va a utilizar es recorrer el mayor número de kilómetros sin repostar, tratando de ir desde cada gasolinera en la que se pare a repostar a la más lejana posible, hasta llegar al destino.

Demostración de optimalidad

Se trata de demostrar que esta estrategia voraz conduce a una solución óptima.

Sea X la solución de la estrategia voraz anterior y sean x_1, x_2, \dots, x_s , las gasolineras en las que el algoritmo decide parar a repostar en dicha solución. Sea Y otra solución compuesta por otro conjunto de gasolineras y_1, y_2, \dots, y_t . Sea N el número total de kilómetros entre el punto de recogida y el punto de entrega, y sea $K[i]$ la distancia en kilómetros recorrida por el vehículo hasta la gasolinera i , siendo $1 \leq i \leq G - 1$. Se tiene:

$$K[i] = \sum_{k=1}^i DG[k] \text{ y } K[G - 1] = N$$

Se trata de demostrar que $s \leq t$, siendo s el número de gasolineras devueltas por el algoritmo voraz y t el número de gasolineras de cualquier otra solución, ya que lo que se quiere minimizar es el número de paradas. Para ello, bastará con demostrar que $x_k \geq y_k$

para todo k . Como X e Y son dos soluciones distintas, sea k el primer índice tal que $x_k \neq y_k$. Sin pérdida de generalidad podemos suponer que $k = 1$ ya que hasta x_{k-1} los viajes son iguales y en la gasolinera x_{k-1} en ambas soluciones se llenan los depósitos. Siguiendo la estrategia voraz propuesta, si $x_1 \neq y_1$ entonces $x_1 > y_1$ ya que x_1 es la gasolinera más alejada a la que se puede viajar sin repostar. Podríamos decir también que $x_2 \geq y_2$ ya que x_2 es la gasolinera más alejada a la que se puede viajar desde x_1 sin repostar. Para probarlo, supongamos por reducción al absurdo que y_2 fuera estrictamente superior que x_2 . Para que en la solución Y se consiga ir desde y_1 a y_2 es que hay menos de n km. entre ellas: $K[y_2] - K[y_1] < n$, por lo que entonces desde x_1 hasta y_2 también hay menos de n km ya que $K[y_1] < K[x_1]$. En este caso, el algoritmo no hubiera escogido x_2 como siguiente gasolinera a x_1 sino y_2 , ya que busca la gasolinera más alejada entre las que puede llegar.

Se repite el proceso y se va obteniendo que $x_k \geq y_k$ para todo k , hasta llegar a la ciudad destino, lo que demuestra que la estrategia voraz consigue una solución óptima.

Algoritmo

Esta estrategia se puede implementar con el siguiente algoritmo, que devuelve un vector de tipo booleano indicando en qué gasolineras debe pararse el vehículo.

```

tipo Vector = matriz[0..G-1] de booleano
fun EntregaExpresMinimasParadas (DG: Vector[1..G-1] de natural, n,G: natural):Vector
    var
        solucion: Vector
    fvar
        para i = 1 hasta G-1 hacer
            solucion[i] ← falso
        fpara
        i ← 0
        contKm ← 0
        repetir
            repetir
                i ← i + 1
                contKm ← contKm + DG[i]
            hasta (contKm > n) ∨ (i = G-1)
            si contKm > n entonces
                i ← i - 1
                solucion[i] ← cierto
                contKm ← 0
            fsi
        hasta i = G-1
        dev solucion[]
    ffun
```

Coste

El coste del primer bucle para está en el orden del número de gasolineras. Lo mismo ocurre con los otros dos bucles, que permiten acceder a cada gasolinera una vez o dos como máximo si se ha excedido en el valor de n al contar los kilómetros. Por lo tanto este algoritmo está en $O(G)$ siendo G el número de gasolineras.

3.8 Problema del robot desplazándose en un circuito

Sea un robot R que dispone de una batería de N unidades de energía. Desde un punto origen se tiene que desplazar hasta el punto de salida S del circuito. En el camino se puede encontrar con obstáculos, O , que serán infranqueables. El paso por una casilla franqueable supone un gasto de energía igual al valor que indica la casilla. Se busca un algoritmo que permita al robot llegar al punto S gastando el mínimo de energía. El circuito se puede representar mediante una matriz en la que desde cada elemento se puede acceder a un elemento adyacente con el consumo de energía que indique la casilla. Veamos el siguiente ejemplo de circuito:

| O | S | O | 2 | 1 |
|---|---|---|---|---|
| 3 | 1 | 3 | 1 | 1 |
| 1 | 6 | 6 | O | 6 |
| 1 | 2 | O | R | 4 |
| 7 | 1 | 1 | 2 | 6 |

El robot se encuentra en la casilla R y tiene que desplazarse con el menor coste de energía posible hasta la casilla S . Desde donde está el robot, se puede desplazar a la derecha consumiendo 4 unidades de energía, abajo consumiendo 2 o en diagonal consumiendo 6 o 1 dependiendo de a qué diagonal se desplace. Donde no se puede desplazar es arriba y a la izquierda, ya que hay obstáculos infranqueables. El algoritmo debería encontrar la ruta más económica en consumo energético desde la casilla R a la casilla S .

El tablero o circuito se puede modelar como un grafo en el que cada casilla es un nodo y el contenido de la casilla el coste energético de acceder a dicho nodo por alguna de sus aristas incidentes. Si representamos el grafo mediante una matriz de adyacencia, el circuito del ejemplo quedaría representado con la matriz de la tabla 3.1. Nótese que el grafo es dirigido y por lo tanto la matriz no es simétrica.

Los nodos se han numerado del 1 al 25 empezando por la posición (1,1) del tablero, siguiendo por la (1,2), (1,3), ... hasta la (5,5). El nodo de la posición (1,1), el 1, es un obstáculo, un nodo al que no se puede acceder, y por lo tanto no tiene aristas con ningún otro nodo, lo que se representa mediante ∞ . Desde el nodo S , el 2, que ocupa en el tablero la posición (1,2), se puede acceder directamente únicamente a los nodos 6,7,8,

que representan las posiciones (2,1), (2,2) y (2,3) respectivamente. Dada una posición (i, j) del tablero, le corresponde el número de nodo $(i - 1) \times 5 + j$. La posición del tablero en el que está el robot, la (4,4), corresponde al nodo número 19, y el coste de acceder a ella desde un nodo que no sea un obstáculo es 0, ya que no se indica otro posible coste, lo mismo ocurre con la posición S .

Con esta representación, el problema se reduce a encontrar un camino de coste mínimo desde la posición en la que se encuentra el robot, R , hasta la posición S . Este problema se puede solucionar utilizando el algoritmo de Dijkstra para calcular la longitud del camino mínimo que va desde el origen hasta el resto de los nodos del grafo. En este caso, el algoritmo Dijkstra se detendrá una vez que el camino hasta el nodo S ya se haya calculado, obteniendo así la ruta de coste mínimo en el camino de R a S .

El algoritmo de Dijkstra adaptado a este problema es el siguiente:

```

tipo VectorNat = matriz[0..n] de natural
fun Dijkstra ( $G = \langle N, A \rangle$ : grafo): VectorNat, VectorNat
  var
    especial, predecesor: VectorNat
    C: conjunto de nodos
  fvar
    Iniciar conjunto C con todos los nodos 1,2,3,..n excepto el 19
    para i  $\leftarrow$  1 hasta n  $\wedge$  i  $\neq$  19 hacer
      especial[i]  $\leftarrow$  Distancia(19,i)
      predecesor[i]  $\leftarrow$  19
    fpara
    mientras C contenga al nodo S hacer {el nodo 2 es S }
      v  $\leftarrow$  nodo  $\in$  C que minimiza especial[v]
      C  $\leftarrow$  C \{v}
      si v  $\neq$  S entonces
        para cada w  $\in$  C hacer
          si especial[w]  $>$  especial[v] + Distancia(v,w) entonces
            especial[w]  $\leftarrow$  especial[v] + Distancia(v,w)
            predecesor[w]  $\leftarrow$  v
        fsi
      fpara
      fsi
    fmientras
    dev especial[],predecesor[]
  ffun

```

El nodo origen será el nodo 19, y el nodo destino el nodo 2. Como el grafo resultante es un grafo disperso, se utilizará un montículo de mínimos que contendrá los nodos de los

que aún no se ha determinado su camino mínimo desde el origen. El valor de ordenación del montículo es la energía consumida desde el origen.

La traza de este algoritmo con la matriz de adyacencia del circuito se puede ver en tabla 3.2. De los nodos que están en C , el valor mínimo de $\text{especial}[]$ se representa en cada paso subrayado, al igual que su valor correspondiente en $\text{predecesor}[]$. Los valores de $\text{especial}[]$ y $\text{predecesor}[]$ de los nodos que no están en C se representan en cursiva. Los valores en dichos arrays correspondientes al nodo R se representan con un carácter guión. En nueve pasos el nodo 2, S , deja de estar en C por lo que ya se ha calculado su camino de coste mínimo desde el origen y el algoritmo finaliza. Nótese que en esta versión del algoritmo se calcula el camino de coste mínimo, sin tener en cuenta el número de aristas. El camino de coste mínimo que encuentra el algoritmo está formado por los siguientes movimientos del robot: primero a la casilla 23, luego a la 22, después a la 16, la 11, la 7 y finalmente alcanza la salida con un gasto de energía de 5.

La demostración de optimalidad y el estudio del coste se puede encontrar en el apartado 3.2.3.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 |
|----|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
| 1 | - | ∞ | |
| 2 | ∞ | - | ∞ | ∞ | ∞ | 3 | 1 | 3 | ∞ |
| 3 | ∞ | ∞ | - | ∞ | |
| 4 | ∞ | ∞ | ∞ | - | 1 | ∞ | ∞ | 3 | 1 | 1 | ∞ | |
| 5 | ∞ | ∞ | ∞ | 2 | - | ∞ | ∞ | 1 | 1 | ∞ | |
| 6 | ∞ | 0 | ∞ | ∞ | ∞ | - | 1 | ∞ | ∞ | ∞ | 1 | 6 | ∞ | |
| 7 | ∞ | 0 | ∞ | ∞ | ∞ | 3 | - | 3 | ∞ | ∞ | 1 | 6 | 6 | ∞ | |
| 8 | ∞ | 0 | ∞ | 2 | ∞ | ∞ | 1 | - | 1 | ∞ | ∞ | 6 | 6 | ∞ | |
| 9 | ∞ | ∞ | ∞ | 2 | 1 | ∞ | ∞ | 3 | - | 1 | ∞ | ∞ | 6 | ∞ | 6 | ∞ | |
| 10 | ∞ | ∞ | ∞ | 2 | 1 | ∞ | ∞ | ∞ | 1 | - | ∞ | ∞ | ∞ | 6 | ∞ | |
| 11 | ∞ | ∞ | ∞ | ∞ | ∞ | 3 | 1 | ∞ | ∞ | ∞ | - | 6 | ∞ | ∞ | ∞ | 1 | 2 | ∞ | |
| 12 | ∞ | ∞ | ∞ | ∞ | ∞ | 3 | 1 | 3 | ∞ | ∞ | 1 | - | 6 | ∞ | ∞ | 1 | 2 | ∞ | |
| 13 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | 1 | 3 | 1 | ∞ | ∞ | 6 | - | ∞ | ∞ | 2 | ∞ | 0 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | |
| 14 | ∞ | - | ∞ | | |
| 15 | ∞ | 1 | 1 | ∞ | ∞ | ∞ | - | ∞ | ∞ | ∞ | 0 | 4 | ∞ | ∞ | ∞ | ∞ | ∞ | |
| 16 | ∞ | 1 | 6 | ∞ | ∞ | ∞ | - | 2 | ∞ | ∞ | ∞ | 7 | 1 | ∞ | ∞ | |
| 17 | ∞ | 1 | 6 | 6 | ∞ | ∞ | 1 | - | ∞ | ∞ | ∞ | 7 | 1 | 1 | ∞ | |
| 18 | ∞ | - | ∞ | |
| 19 | ∞ | 6 | ∞ | 6 | ∞ | ∞ | ∞ | - | 4 | ∞ | ∞ | 1 | 2 | 6 |
| 20 | ∞ | 6 | ∞ | ∞ | ∞ | 0 | - | ∞ | ∞ | ∞ | 2 | 6 |
| 21 | ∞ | 1 | 2 | ∞ | ∞ | ∞ | - | 1 | ∞ | ∞ | |
| 22 | ∞ | 1 | 2 | ∞ | ∞ | ∞ | 7 | - | 1 | ∞ | ∞ |
| 23 | ∞ | 2 | ∞ | 0 | ∞ | ∞ | 1 | - | 2 | ∞ |
| 24 | ∞ | 0 | 4 | ∞ | ∞ | 1 | - | 6 |
| 25 | ∞ | 0 | 4 | ∞ | ∞ | ∞ | 2 | - |

Tabla 3.1: Matriz de adyacencia del circuito del robot

Tabla 3.2: Trazo de la particularización del algoritmo de Dijkstra al circuito del robot

3.9 Asistencia a incidencias

Una empresa de software debe atender las incidencias de sus clientes con la máxima celeridad posible. Cada jornada laboral se planifican n salidas sabiendo de antemano el tiempo que va a llevar atender cada una de las incidencias. Así, la incidencia i -ésima tardará t_i minutos. La empresa quiere dar el mejor servicio posible a sus clientes y uno de sus criterios es que éstos esperen lo menos posible a que se resuelva su problema. Para ello la empresa pretende minimizar el tiempo medio de espera de sus clientes.

Vamos a estudiar cómo se aplica el esquema voraz para resolver este problema en los siguientes casos:

1. La empresa sólo dispone de un experto para resolver las incidencias.
2. La empresa dispone de E expertos para resolver las incidencias.

Empecemos con el caso 1. Este problema es un ejemplo de “Minimización del tiempo en el sistema” descrito en el apartado 3.3.1. En él se indica que, como n es conocido, minimizar el tiempo medio de espera de los n clientes equivale a minimizar el tiempo total que están en el sistema todos los clientes. Si TE_i es el tiempo de espera del cliente i -ésimo, se trata entonces de minimizar la expresión:

$$TE(n) = \sum_{i=1}^n TE_i$$

Por lo tanto, en este caso se puede utilizar el esquema voraz ya que existe una función de selección que garantiza obtener una solución óptima. La función de selección consiste en atender a los clientes en orden no decreciente de sus tiempos de atención. La demostración de optimalidad y el coste computacional se pueden encontrar en el apartado 3.3.1.

En cuanto al caso 2, también se puede utilizar el esquema voraz ya que el objetivo es el mismo. En este caso se disponen de E expertos para atender a los n clientes, por lo que existen E expertos dando servicio simultáneamente. La forma óptima de atender a los clientes es la siguiente:

1. Se ordenan los avisos por orden no decreciente de tiempo de reparación.
2. Se asignan los avisos por ese orden siempre al experto menos ocupado. En caso de haber varios con el mismo grado de ocupación, se escoge el de número menor. Así, si los avisos están ordenados de forma que $t_i \leq t_j$ si $i < j$, asignaremos al experto k los avisos $k, k+E, k+2E, \dots$

3.10 Ejercicios propuestos

1. ¿Puede un grafo tener dos árboles de recubrimiento mínimo diferentes? Se pide, en caso afirmativo, poner un ejemplo y, en caso negativo, justificar la respuesta.
2. Considere el problema de la mochila: tenemos una mochila de capacidad M , n objetos con beneficios $b_1, b_2, b_3, \dots, b_n$ y pesos $p_1, p_2, p_3, \dots, p_n$. El objetivo es maximizar el valor de los objetos transportados, respetando la limitación de la capacidad impuesta M . Se pide indicar si este problema se podría resolver de forma óptima con el esquema voraz en el caso de que cada objeto pueda meterse en la mochila, no meterse, o meter la mitad del objeto obteniendo la mitad del beneficio.
3. Considere el problema de la devolución de cambio de monedas. Supongamos que disponemos de un conjunto finito de tipos de moneda $M = \{m_0 < m_1 < m_2 < \dots < m_n\}$, que $m_0 = 1$ y que $m_i = v_{i-1} \times m_{i-1}$ para todo i entre 1 y n siendo $v_{i-1} > 1$. Se pide demostrar que la estrategia voraz de seleccionar los tipos de moneda de mayor a menor, cogiendo tantas unidades como sea posible de cada tipo hasta llegar a la cantidad deseada C , da lugar a una solución óptima.
4. Un dentista pretende dar servicio a n pacientes y conoce el tiempo requerido por cada uno de ellos, siendo t_i , $i = 1, 2, \dots, n$, el tiempo requerido por el paciente i -ésimo. Como todos los pacientes llegan cuando se abre la consulta, el objetivo es minimizar el tiempo total que todos los pacientes están en la sala de espera. Se pide identificar, si la hubiera, una función de selección que garantice que un algoritmo voraz puede construir una planificación óptima. También habrá que demostrar la optimalidad de dicha función.
5. Dado un conjunto $\{r_1, r_2, \dots, r_n\}$ de puntos de la recta real, determine el menor conjunto de intervalos cerrados de longitud 1 que contenga a todos los puntos dados.
6. Se pide demostrar lo siguiente o dar un contraejemplo: para el problema de “Minimización del tiempo en el sistema” la planificación de clientes por orden descendente de tiempos de servicio da lugar a la peor planificación posible.
7. Tras unas lluvias torrenciales, las calles de una ciudad han quedado seriamente dañadas. La institución competente no puede arreglar todas sus calles debido al elevado coste que ello supondría, por lo que han decidido volver a pavimentar sólo aquellas que les permitan ir de una intersección a otra cualquiera de la ciudad. Quieren gastarse lo menos posible en la pavimentación, teniendo en cuenta que el coste es directamente proporcional a la longitud de las calles que hay que pavimentar. Se pide proponer un algoritmo que resuelva de forma óptima este problema, demostrando la optimalidad de la solución e indicando su coste.

8. Un hortelano posee n huertas, cada una con un tipo diferente de árbol frutal. Las frutas han madurado y es hora de recolectarlas. La recolección de una huerta dura un día completo. Se sabe, para cada huerta, el beneficio de lo que obtendría por la venta de lo recolectado. También se sabe los días que tardan en pudrirse los frutos de cada huerta. Se pide:
 - (a) Proponer una estrategia voraz que ayude al hortelano a decidir qué debe recolectar y cuándo debe hacerlo de forma que maximice el beneficio.
 - (b) Estudiar si la estrategia utilizada para solucionar el apartado anterior es válida también en el caso de que la recolección de cada huerta requiera un número arbitrario de días.
9. Se dispone de un conjunto de ficheros f_1, f_2, \dots, f_n con tamaños l_1, l_2, \dots, l_n y de un disquete con una capacidad total de almacenamiento de D , siendo $D < l_1 + l_2 + \dots + l_n$. Se pide:
 - (a) Suponiendo que se desea maximizar el número de ficheros almacenados y que se hace uso de una estrategia voraz basada en la selección de ficheros de menor a mayor tamaño, indicar si dicha estrategia siempre obtendría una solución óptima. En caso afirmativo se pide demostrar la optimalidad y en caso negativo poner un contraejemplo.
 - (b) En el caso de que quisiéramos ocupar la mayor cantidad de espacio en el disquete independientemente del número de ficheros almacenados, ¿una estrategia voraz basada en la selección de ficheros de mayor a menor tamaño obtendría en todos los casos la solución óptima? En caso afirmativo se pide demostrar la optimalidad y en caso negativo poner un contraejemplo.
10. Se desean asignar n tareas a n trabajadores de forma que cada uno realice una tarea y se maximice el valor total de ejecutar las n tareas. Sea $v_{ij} > 0$ el rendimiento de que el trabajador i realice la tarea j . La asignación de trabajadores a tareas corresponde a la asignación de valores 0 o 1 a las variables x_{ij} , siendo $1 \leq i, j \leq n$. Si $x_{ij} = 1$ significa que al trabajador i se le ha asignado la tarea j , y $x_{ij} = 0$ significa que al trabajador i no se le ha asignado la tarea j . El valor de una asignación es $\sum_i \sum_j v_{ij} x_{ij}$. Se pide escribir dos algoritmos para dos funciones diferentes de selección. Una función asigna a un trabajador el mejor trabajo posible. La otra asigna a un trabajo el mejor trabajador posible. Demuestra que ninguna de las funciones conduce a una solución óptima. Indica si una de las funciones es siempre mejor que la otra.

3.11 Notas bibliográficas

[HS94] y [BB06] son dos referencias básicas en algoritmia y por lo tanto también a la hora de estudiar los algoritmos voraces. En [CLRS01] se pueden encontrar los funda-

mentos teóricos de los métodos voraces. La demostración de optimalidad del problema del cambio de moneda está tomada de [Gue00]. El problema 3.7 está inspirado en el problema del “Camionero con prisa” de [Gue00]. En este libro se pueden encontrar otros problemas voraces resueltos, al igual que en [MOOMV03]. En ambos se hallan las soluciones de varios de los problemas propuestos en este capítulo. En [GA97] también se pueden encontrar problemas resueltos.

En la Web se pueden encontrar animaciones de varios algoritmos voraces, como los de Prim, Kruskal y Dijkstra, que pueden facilitar su comprensión.

Capítulo 4

Divide y vencerás

4.1 Planteamiento y esquema general

La estrategia de *Divide y Vencerás* es una técnica algorítmica que **se basa en la descomposición de un problema en subproblemas de su mismo tipo**, lo que permite disminuir la complejidad y en algunos casos, paralelizar la resolución de los mismos.

La estrategia del algoritmo es la siguiente:

- Descomposición del problema en subproblemas de su mismo tipo o naturaleza.
- Resolución recursiva de los subproblemas.
- Combinación, si procede, de las soluciones de los subproblemas.

Desde un punto de vista algorítmico, en la primera de las fases se determina el número y tamaño de los subproblemas, siendo éste uno de los pasos determinantes para la complejidad posterior del algoritmo. Seguidamente se realizan llamadas recursivas al algoritmo para cada uno de los subproblemas. Por último, si es necesario, se combinan las soluciones parciales y se obtiene la solución del problema.

Este esquema aplica el principio de inducción sobre los diversos ejemplares del problema; de esta manera supone solucionados los subproblemas, y utiliza las soluciones parciales de los mismos para componer la solución al problema. Para casos suficientemente pequeños, el esquema proporciona una solución trivial no recursiva, lo que equivale a la base de la inducción.

El ejemplo conocido más antiguo de aplicación de una técnica recursiva similar es el del algoritmo de Euclides (hacia el 300 a.C.) que calcula el Máximo Común Divisor de dos enteros. Otras aplicaciones no algorítmicas conocidas eran las usadas también en procesos de ordenación y clasificación. El esquema, desde el punto de vista algorítmico, no aparece hasta 1946 en un artículo publicado por John Mauchly en el que acuñaba el término *Divide and conquer* para identificar a esta técnica de descomposición algorítmica [Knu73].

Un ejemplo sencillo de aplicación de la técnica de *Divide y Vencerás* es la búsqueda binaria en un vector ordenado. Dado un vector $v[1..n]$ de elementos ordenados, se puede verificar la pertenencia de un valor x a dicho vector mediante el siguiente algoritmo:

```
fun Bbinaria(i,j:entero;v: vector [1..N] de entero; x:entero): booleano
  var
    m:entero
  var
    si i = j entonces
      si v[i] = x entonces
        dev verdadero
      sino
        dev falso
      fsi
    sino
      m  $\leftarrow$  (i+j) div 2
      si v[m]  $\leq$  x entonces
        bbinaria(i,m,v,x)
      sino
        bbinaria(m+1,j,v,x)
      fsi
  ffun
```

La llamada inicial incluye el vector completo:

Bbinaria(1,n,v,x)

El algoritmo comprueba en primer lugar si el problema es lo suficientemente pequeño como para poder dar una solución inmediata sin necesidad de volver a descomponerlo. En este caso con vectores de tamaño 1, el problema es resoluble de manera trivial.

En caso contrario el problema exige una descomposición del vector en dos subvectores, eligiendo proseguir la búsqueda en uno de ellos. Por ejemplo, sea el vector $v = (1, 3, 8, 12, 13, 32, 56)$ y buscamos el elemento $x = 32$.

La aplicación del algoritmo genera la siguiente pila de invocaciones:

Bbinaria(1,7,(1,3,8,12,13,32,56),32) con $m = 4$
 Bbinaria(5,7,(-,-,-,-,13,32,56),32) con $m = 6$
 Bbinaria(5,6,(-,-,-,-,13,32,-),32),x
 Bbinaria(1,n,v,x)

Visto este ejemplo, podemos dar ya un *esquema general* de la técnica de *divide y vencerás*:

```
fun DyV(problema)
    si trivial(problema) entonces
        dev solución-trivial
    sino hacer
         $\{p_1, p_2, \dots, p_k\} \leftarrow$  descomponer(problema)
        para  $i \in (1..k)$  hacer
             $s_i \leftarrow$  DyV( $p_i$ )
        fpara
        fsi
        dev combinar( $s_1, s_2, \dots, s_k$ )
ffun
```

Los elementos del esquema son los siguientes:

Trivial y solución-trivial: un problema es trivial si la solución puede darse sin necesidad de descomponer el problema. En tal caso, una función se encarga de dar la solución a un problema de tamaño suficientemente pequeño.

Descomponer: realiza la división del problema en subproblemas. La descomposición exige que los subproblemas sean de menor tamaño que el problema inicial. El tipo de sucesión formada por los sucesivos tamaños del problema y el número de subproblemas determinan la complejidad algorítmica.

Combinar: aplicando el principio de inducción, se dan los subproblemas por resueltos. En estas condiciones, lo que queda por abordar desde el punto de vista algorítmico es cómo realizar (si es el caso) la combinación de las soluciones de los subproblemas para obtener la solución del problema final. Si no hay que realizar combinación de soluciones el tipo de problema se conoce como *reducción*.

La técnica *divide y vencerás* se basa en una extensión del principio de inducción, que nos indica que para probar un aserto general (válido para cualquier número natural) éste puede ser demostrado para un caso particular de tamaño suficientemente pequeño y luego, supuesto cierto para tamaño k , demostrar que lo es para $k + 1$. En el caso de la técnica algorítmica, el problema se descompone en uno o más subproblemas y se suponen resueltos por separado, para posteriormente combinar las soluciones parciales en una solución global.

La principal dificultad de este tipo de algoritmos está en determinar cómo combinar las soluciones. Generalmente se tiende a razonar su solución haciendo hincapié en las sucesivas recursiones que llevan al problema trivial. En general para una estrategia de

resolución hay que plantearse más bien cómo componer una solución al problema inicial suponiendo conocidas las soluciones a los problemas parciales. Por último, se aborda el caso trivial.

El coste de este tipo de algoritmos depende de cuántas descomposiciones se realicen y del tipo de decrecimiento de las sucesivas llamadas recursivas a los subproblemas. En general se plantea una ecuación de recurrencia del tipo:

$$T(n) = aT(n/b) + cn^k$$

en el caso de un decrecimiento del tamaño del problema en progresión geométrica. O bien del tipo:

$$T(n) = aT(n - b) + cn^k$$

para decrecimientos en progresión aritmética.

En ambos casos a es el número de subproblemas en los que se descompone el problema inicial, b es el factor de reducción del tamaño y la expresión polinómica indica el coste de la función de combinación de las soluciones parciales. Si $k = 0$ hablamos de un problema de *reducción*. Es el caso del algoritmo de la búsqueda binaria que acabamos de exponer.

Las ecuaciones de recurrencia del tipo:

$$T(n) = \begin{cases} cn^k & , \text{ si } 1 \leq n < b \\ aT(n/b) + cn^k & , \text{ si } n \geq b \end{cases}$$

tienen la siguiente resolución:

$$T(n) \in \begin{cases} \Theta(n^k) & , \text{ si } a < b^k \\ \Theta(n^k \log n) & , \text{ si } a = b^k \\ \Theta(n^{\log_b a}) & , \text{ si } a > b^k \end{cases}$$

y las del tipo:

$$T(n) = \begin{cases} cn^k & , \text{ si } 1 \leq n < b \\ aT(n - b) + cn^k & , \text{ si } n \geq b \end{cases}$$

tienen la siguiente resolución:

$$T(n) \in \begin{cases} \Theta(n^k) & , \text{ si } a < 1 \\ \Theta(n^{k+1}) & , \text{ si } a = 1 \\ \Theta(a^{n/b}) & , \text{ si } a > 1 \end{cases}$$

4.2 Ordenación por fusión (*Mergesort*)

La ordenación por fusión es otro ejemplo sencillo para entender la técnica de *divide y vencerás*. Dado un vector de enteros, lo que plantea es dividir el vector por la mitad e invocar al algoritmo para ordenar cada mitad por separado. Tras ésta operación, sólo queda fusionar esos dos subvectores ordenados en uno sólo, también ordenado.

La fusión (*merge*) se hace tomando, sucesivamente, el menor elemento de entre los que queden en cada uno de los dos subvectores.

El algoritmo de ordenación *Mergesort* queda como sigue:

```
fun Mergesort (T: vector [1..n] de entero): vector [1..n] de entero
    var
        U: vector [1..n] de entero, V: vector [1..n] de entero
    fvar
        si trivial(n) entonces Insertar(T[1..n])
        sino
            U[1..1 + ⌊n / 2⌋] ← T[1..⌊n / 2⌋]
            V[1..1 + ⌈n / 2⌉] ← T[1 + ⌈n / 2⌉..n]
            Mergesort(U)
            Mergesort(V)
            Fusionar(U,V,T)
    fsi
```

ffun

Se utiliza el algoritmo de ordenación por inserción para los tamaños pequeños. La combinación de las soluciones parciales es la mezcla o fusión de los subvectores ordenados:

```
fun Fusionar (U:vector [1..n+1] de entero, V:vector [1..m+1] de entero, T:vector [1..m+n] de entero)
```

```
    var
        i,j: natural
    fvar
        i,j ← 1
        U[m+1],V[n+1] ← ∞
    para k ← 1 hasta m+n hacer
        si U[i] < V[j]
        entonces T[k] ← U[i]
            i ← i + 1
        sino T[k] ← V[j]
            j ← j + 1
    fsi
    fpara
```

ffun

Puesto que tanto la separación en dos subvectores como su posterior fusión tiene coste lineal, la ecuación de recurrencia es:

$$T(n) = 2T(n/2) + cn$$

con $a = b = 2$ y $k = 0$, es decir, $t(n) \in \Theta(n \log n)$.

4.3 El puzzle *tromino*

Un tromino es una figura geométrica compuesta por 3 cuadros de tamaño 1×1 en forma de L . Sobre una retícula cuadrada de $n \times n$ (siendo $n = 2^k$) se dispone un cuadrado marcado (en negro, por ejemplo) de tamaño 1×1 y el resto vacío. El problema consiste en llenar el resto de la cuadrícula de trominos sin solaparlos y cubriéndola totalmente, exceptuando el cuadrado mencionado. Aunque en una primera aproximación puede pensarse que se trata de un problema resoluble únicamente mediante exploración ciega y la técnica de vuelta atrás, hay una manera de realizarlo mediante *divide y vencerás* si se cumple la condición de que n sea potencia de 2.

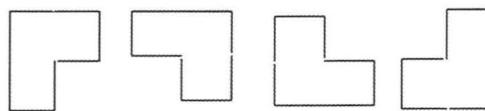


Figura 4.1: Rotaciones de un *tromino*

De esta manera, supongamos que tenemos que resolver el problema para un tamaño 8×8 . Partimos de una cuadrícula vacía y con una de las casillas señalada en negro. Esta casilla no podrá usarse ni solaparse con trominos. El resto de las casillas deben llenarse con trominos en cualquiera de las orientaciones posibles, como se muestra en la figura 4.1. Se supone que se dispone de un numero ilimitado de ellos (no hay limitaciones respecto al número de veces que se usa cada orientación).

Para abordar el problema partimos de la demostración de que es posible dividir el resto de las cuadrículas en grupos de 3, es decir, que $n^2 - 1$ es múltiplo de 3, con $n = 2^k$. Vemos que para $k = 1$ se cumple. Por inducción, suponemos cierto que $2^{2k} - 1$ es múltiplo de 3, es decir que $2^{2k} - 1 = 3p$ con p un valor natural. Veamos ahora el paso de inducción:

$$2^{2(k+1)} - 1 = 2^{2k+2} - 1 = 2^2 2^{2k} - 1 = 4(3p + 1) - 1 = 12p + 3 = 3(4p - 1)$$

Pasamos a razonar ahora sobre el algoritmo. La manera de resolver el problema es dividir el cuadrado en 4 cuadrantes. Aquel que tiene la casilla negra (casilla marcada) queda

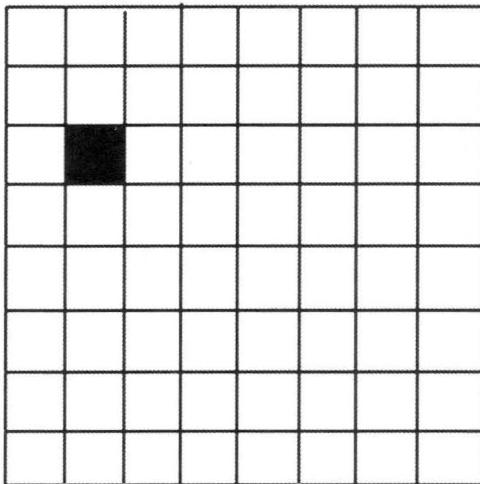


Figura 4.2: Retículo con casilla en negro

como está. A continuación ubicamos un tromino en el centro del tablero de manera que esté orientado hacia el cuadrante con la casilla negra. Cada uno de los 3 cuadrantes que intersecan con el tromino usarán como casilla marcada la intersección con el mismo (figura 4.3). Seguidamente para cada uno de los cuadrantes, se llama recursivamente a la función.

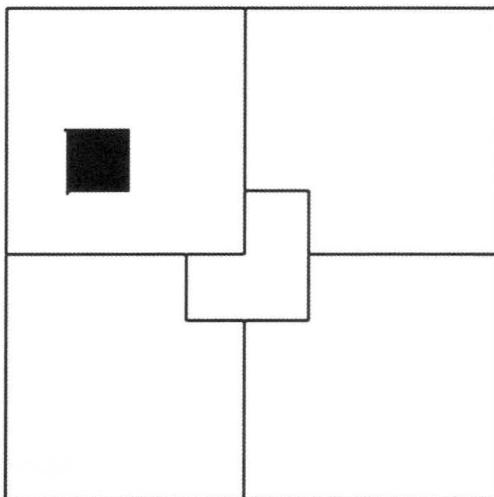


Figura 4.3: Proceso de resolución en primera recursión

Los elementos del esquema aplicados al problema son los siguientes:

Caso trivial: para el caso de que $n = 2$ podremos dar una solución si una de las casillas es la casilla en negro o está *marcada*. En este caso elegimos una de las rotaciones posibles del tromino y lo colocamos completando la retícula de 2×2 .

Descomponer: los cuadrados de tamaño $2^k \times 2^k$ se descomponen en 4 cuadrículas $2^{k-1} \times 2^{k-1}$. Cada uno de ellos tendrá una casilla *marcada*. Una de estas casillas será la casilla en negro inicial. Las otras 3 casillas marcadas serán las 3 casillas del tromino colocado en el centro del tablero, como indica la figura 4.3

Combinar: se recogen en el cuadrado solución los triminos ubicados en cada uno de los cuadrantes.

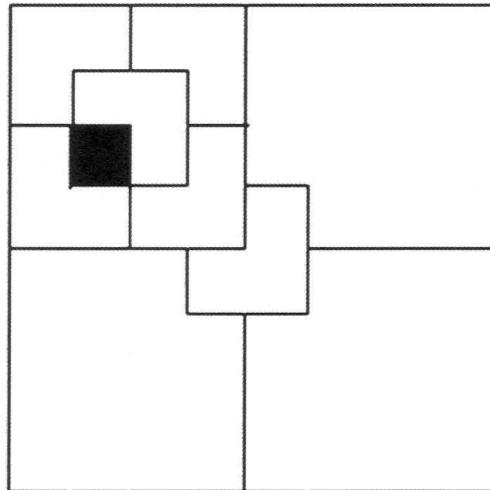


Figura 4.4: Proceso de resolución en segunda recursión

Consideremos los parámetros del algoritmo, que debe contener un tablero T , el tamaño del mismo, n y la posición de la casilla *marcada*, que denominaremos m . El algoritmo queda como sigue:

```

fun Tromino( $T$ :matriz [1..n,1..n] de natural,n,m:natural)
  var
     $T_1, T_2, T_3, T_4$  :matriz [1..n,1..n] de natural
  fvar

  si  $n = 2$  entonces
     $T \leftarrow$  colocaTromino( $T, m$ )
  
```

```

dev T
sino
   $m' \leftarrow$  esquina cuadrante con casilla negra
  colocaTromino( $T, m'$ )
   $T_1, T_2, T_3, T_4 \leftarrow$  dividir  $T$  en 4 cuadrantes
  tromino( $T_1, n/2, m_1$ )
  tromino( $T_2, n/2, m_2$ )
  tromino( $T_3, n/2, m_3$ )
  tromino( $T_4, n/2, m_4$ )
fsi
   $T \leftarrow$  combinar( $T_1, T_2, T_3, T_4$ )
dev T
ffun

```

El coste del algoritmo se calcula a partir de la expresión de $T(n) = 4T(n/2) + c$, con $b = 2$, $a = 4$ y $k = 0$. Se suponen unitarias la descomposición de T en subcuadrantes y la superposición de éstos en T . De no ser así, podría formularse el algoritmo parametrizando cuadrantes directamente en la variable T . El resultado de la ecuación cuando $a > b^k$ es $\Theta(n^{\log_b a})$, es decir $\Theta(n^2)$ u orden cuadrático.

4.4 Ordenación rápida (*Quicksort*)

En la ordenación por fusión se pone el énfasis en cómo combinar los subproblemas resueltos, mientras que la operación de descomponer en subproblemas es trivial.

Sin embargo, si al descomponer el vector, los elementos del primer subvector fueran todos menores o iguales que los del segundo subvector, no habría que realizar ninguna operación de combinación. En este caso, la descomposición es algo más compleja, pero la combinación es trivial.

Esta es la idea que subyace al algoritmo *quicksort*, presentado por primera vez en [Hoa62] y que realiza el siguiente proceso:

- Toma un elemento cualquiera del vector denominado *pivote*.
- Toma los valores del vector que son menores que el pivote y forma un subvector. Se procede análogamente con los valores mayores o iguales.
- Se invoca recursivamente al algoritmo para cada subvector.

Para realizar la descomposición, se suele elegir como pivote el primer elemento del vector. Para almacenar los subvectores, se puede usar el propio vector situando los menores que el pivote a su izquierda, y los mayores a su derecha. Una forma eficiente de realizar este proceso es comenzando a recorrer el vector por ambos extremos. En

el recorrido de izquierda a derecha, nos detenemos cuando se encuentre un elemento mayor que el pivote. En el recorrido de derecha izquierda, nos detenemos al encontrar un elemento menor que el pivote. Entonces se intercambian ambos elementos, y se prosigue el recorrido. Cuando los recorridos se cruzan, el vector está descompuesto satisfactoriamente:

```
fun Pivatar (T:vector [i..j] de entero)
  var
    p,k:entero
  fvar
    p  $\leftarrow T[i]
    k  $\leftarrow i; l \leftarrow j + 1
    repetir k  $\leftarrow k + 1$  hasta T[k]  $> p \vee k \geq j
    repetir l  $\leftarrow l - 1$  hasta T[l]  $\leq p$ 
    mientras k < l hacer
      intercambiar(T,k,l)
      repetir k  $\leftarrow k + 1$  hasta T[k]  $> p
      repetir l  $\leftarrow l - 1$  hasta T[l]  $\leq p$ 
    fmientras
    intercambiar(T,i,l)
ffun$$$$ 
```

Por ejemplo, dado el vector [5,6,9,3,4,1] el resultado de pivotar sobre el primer elemento del vector (el valor 5 en negrita) compondrá el vector [3,4,1,**5**,6,9]. Una vez establecido el procedimiento de pivote, el algoritmo es muy sencillo:

```
fun Quicksort (T[i..j])
  si trivial(j-i) entonces Insertar(T[i..j])
  sino
    Pivotar(T[i..j],l);
    Quicksort(T[i..l-1]);
    Quicksort(T[l+1..j])
  fsi
ffun
```

Para que sea eficiente, la descomposición en subproblemas tiene que ser equilibrada. En el caso peor en el que el pivote fuera el menor (o el mayor) valor del vector generaríamos un vector de un solo elemento y otro vector con los demás. En ese caso, necesitamos $O(n)$ llamadas recursivas, y como *Pivotar* tiene un coste lineal, el algoritmo es de orden $O(n^2)$. A pesar de esta aparente contrariedad, el algoritmo es el mejor conocido en el caso promedio. Se puede encontrar una demostración detallada del coste en [BB90], así como técnicas de elección del pivote. De hecho es interesante comentar que si se utiliza como pivote la mediana del vector, o que requiere un tiempo lineal, se puede

pivотar alrededor de ella en tiempo $O(n)$ y entonces el algoritmo *quicksort* mejorado sería $O(n \log n)$ incluso en el caso peor. Sin embargo, la constante oculta se vuelve tan grande que lo hace menos eficiente que otros algoritmos como la ordenación con montículo (ver pág. 42).

Otra elección interesante es el tamaño umbral a partir del cual llamar a otro algoritmo como el de ordenación por inserción. En [Knu73], por ejemplo, se recomienda utilizar otro algoritmo para valores de n por debajo de 9.

4.5 Cálculo del elemento mayoritario en un vector

Dado un vector $v[1..n]$ de números naturales, se quiere averiguar si existe un elemento *mayoritario*, es decir que aparezca al menos $n/2 + 1$ veces en el vector.

Siguiendo los pasos indicados en la exposición del esquema, la descomposición puede ser en 2 subproblemas de tamaño $n/2$, que es la más eficiente. En cuanto a la combinación de los subproblemas resueltos, el algoritmo nos debe proporcionar bien el valor mayoritario o bien indicarnos que éste no existe. Por ejemplo, en este caso, si se tienen valores naturales en el vector podemos acordar el resultado -1 como indicador de inexistencia de elemento mayoritario, y valores positivos para el elemento mayoritario de ese subvector. Como caso trivial tendremos los vectores de un único elemento, cuyo elemento mayoritario es él mismo.

Partimos del esquema principal y particularizamos:

```
fun Mayoritario(i,j:natural; v:vector [1..n] de natural): entero
  var
    m:natural
    s1,s2:entero
  fvar
  si i = j entonces
    dev v[i]
  sino
    m  $\leftarrow (i + j) \div 2$ 
    s1  $\leftarrow$  Mayoritario(i,m,v)
    s2  $\leftarrow$  Mayoritario(m+1,j,v)
    dev Combinar(s1,s2)
ffun
```

La función de combinación debe recibir los valores s_1 y s_2 y decidir qué implicaciones tienen con respecto del vector inicial. Los casos que se nos pueden presentar son los siguientes:

- s_1 y s_2 valen ambos -1 . En tal caso, incluso si $s_1 = s_2$, no hay elemento mayoritario, ya que al menos una de las dos mitades debe tener uno.

- Uno de los dos valores es -1 y otro no. En este caso un subvector tiene un elemento mayoritario. Para comprobar si es mayoritario en el resto del vector lo que hacemos es comprobar si ocurre $n/2 + 1$ más veces.
- Ambos valores son iguales y no negativos, lo que indica que dicho valor es el elemento mayoritario.
- Ambos valores son no negativos, luego ambos son candidatos a ser mayoritario y habrá que realizar dos comprobaciones.

La función *Combinar* queda por tanto como sigue:

```
fun Combinar (a,b:entero;v:vector [1..n] de natural):entero
  si a = -1  $\wedge$  b = -1 entonces dev -1 fsi
  si a = -1  $\wedge$  b  $\neq$  -1 entonces dev ComprobarMayoritario(b,v) fsi
  si a  $\neq$  -1  $\wedge$  b = -1 entonces dev ComprobarMayoritario(a,v) fsi
  si a  $\neq$  -1  $\wedge$  b  $\neq$  -1 entonces
    si ComprobarMayoritario(a,v) = a entonces dev a
    sino si ComprobarMayoritario(b,v) = b entonces dev b fsi
    fsi
  fsi
```

fun

Por último la función *ComprobarMayoritario* recorre el vector y cuenta las apariciones del argumento para ver si son más de $n/2$; en tal caso devuelve el valor pasado como argumento para su comprobación, y en caso contrario devuelve -1 .

```
fun ComprobarMayoritario (x:natural;v:vector [1..n] de natural):entero
  var
    c:natural
  fvar
    c  $\leftarrow$  1
  para k  $\leftarrow$  1 hasta n hacer
    si v[k]=x entonces c  $\leftarrow$  c + 1 fsi
  fpara
  si c > n entonces dev c sino dev -1 fsi
```

ffun

La función de recurrencia asociada es la siguiente:

$$T(n) = 2T(n/2) + cn$$

ya que en el peor caso, es necesario recorrer el vector tras las llamadas recursivas para contar los elementos mayoritarios. En este caso tenemos $a = b = 2$ y $k = 0$ y por tanto: $t(n) \in \Theta(n \log n)$.

4.6 Liga de equipos

Supongamos que n equipos (con $n = 2^k$) desean realizar una liga. Las condiciones en las que se celebra el torneo son las siguientes: cada equipo puede jugar un partido al día, y la liga debe celebrarse en $n - 1$ días. Damos por hecho que disponen de suficientes campos de juego.

El problema plantea la realización de un calendario en el que por cada par de equipos se nos indique el día en que juega. La solución tendrá por tanto forma de matriz simétrica con $M[e_i, e_j] = d_{ij}$ indica que el equipo e_i juega el día $= d_{ij}$ con el equipo e_j .

| | e_1 | e_2 | ... | e_n |
|-------|-------|-------|-----|-------|
| e_1 | - | 3 | ... | 1 |
| e_2 | 3 | - | ... | 1 |
| ... | ... | ... | ... | ... |
| e_n | 1 | 4 | | - |

En este caso se puede comprobar que podemos realizar una descomposición en subproblemas de tamaño $n/2$ que pueden darse por resueltos en $n/2 - 1$ días. En este punto tenemos por tanto dos conjuntos disjuntos de equipos que ya han jugado entre sí; sean $c_A = \{e_1, e_2, \dots, e_{n/2}\}$ y $c_B = \{e_{n/2+1}, e_{n/2+2}, \dots, e_n\}$.

El resultado del problema se puede almacenar en una tabla $T[1..n, 1..n]$ simétrica como la de la figura anterior, con $T[i, j] = d_{ij}$, siendo d_{ij} el día del partido entre el equipo e_i y el equipo e_j . El algoritmo por tanto necesita tener como argumento, tanto los equipos que juegan como el día en que empieza el torneo.

Para la resolución algorítmica, partimos del esquema principal y particularizamos cada uno de los elementos del esquema:

Caso trivial: la liga consta de sólo dos equipos e_i y e_j y juegan en $n - 1 = 2 - 1 = 1$ días. Es decir, dos equipos que juegan empezando el día d , pueden jugar ese mismo día. El resultado es una entrada en la tabla T que consigne el encuentro entre e_i y e_j el día d de la forma $T[i, j] = d$.

Descomponer: si el rango inicial es $1..n$, la partición sería $[1..n/2]$ y $[n/2 + 1..n]$. Generalizando y usando los parámetros del algoritmo, el torneo se circunscribirá a los equipos del i al j , con lo que si hacemos $m = (i + j - 1)/2$ un rango será del i al m y otro del $m + 1$ al n , ambos con $n/2$ equipos, considerando n como el número de equipos correspondiente al rango $i..j$.

Combinar: para combinar, tenemos que aplicar el principio de inducción sobre los subproblemas y darlos por resueltos. Suponemos que los conjuntos de equipos

$c_A = \{e_1, e_2, \dots, e_{n/2}\}$ y $c_B = \{e_{n/2+1}, e_{n/2+2}, \dots, e_n\}$ han jugado ya todos entre sí y en los primeros $n/2 - 1$ días. A continuación, tenemos que ver la manera de organizar los partidos que faltan en los días que quedan hasta los $n - 1$ de que disponemos. Esto implica que los equipos que todavía no han jugado entre sí deben hacerlo en $n/2$ días. Y ¿quienes faltan por jugar? Pues evidentemente equipos de c_A frente a equipos de c_B , ya que ambos conjuntos tienen intersección nula y en el seno de los mismos se ha completado ya la liga (han jugado ya *todos con todos*).

Consiguientemente, pongamos a los componentes de c_A frente a los de c_B y realicemos el día $n/2$ el primer torneo entre equipos de ambos conjuntos. Este primer día pueden enfrentarse por ejemplo e_1 con $e_{n/2+1}$, e_2 con $e_{n/2+2}$, etc. hasta $e_{n/2}$ con e_n . Al día siguiente desplazamos una posición a los componentes del segundo conjunto para que e_1 ahora juegue con $e_{n/2+2}$, e_2 con $e_{n/2+3}$, $e_{n/2-1}$ con e_n y así sucesivamente.

A partir de este análisis, la realización del algoritmo es sencilla. En primer lugar fijamos el caso trivial y la descomposición en subproblemas con sendas llamadas recursivas a la función. Para el caso de la combinación de las soluciones parciales, llamamos a otra función auxiliar que nos resuelva lo que queda de torneo.

El algoritmo tendrá 3 parámetros: el equipo inicial i , el equipo final j y el día que comienza el torneo (por ejemplo el día $d = 1$). La llamada inicial es:

$$\text{torneo}(1, n, 1)$$

equivalente a organizar un torneo entre los equipos del 1 al n empezando el día 1. Para el caso general, si se organiza la liga comenzando el día d entre los equipos del subconjunto $C = \{i..j\}$, éstos partidos se deberán jugar en $c - 1$ días, con $c = \text{cardinal}(C) = j - i + 1$, y terminar el día $d + c - 2$.

El algoritmo queda como sigue:

fun *Torneo*(i,j,d:natural)

var

 m,n: natural;

fvar

 m \leftarrow (i+j-1)/2;

 n \leftarrow j-i+1;

si $n = 2$ **entonces**

 T[i,j] \leftarrow d;

sino

Torneo(i,m,d)

Torneo(m+1,j,d)

Combinar(i,j,d+1)

fsi**ffun**

La función *combinar* organiza la liga entre los subconjuntos disjuntos c_A y c_B mencionados anteriormente. Como hemos comentado antes, esta función recorrerá el segundo conjunto usando aritmética modular para realizar el desplazamiento y la asignación de partidos. Por cada desplazamiento se consume un día. Hay por tanto dos bucles anidados: uno que realiza el desplazamiento, y otro que recorre los elementos de ambos conjuntos y asigna el día del encuentro entre los mismos.

El día $n/2$ (el día siguiente a la finalización de los subproblemas) comenzaríamos los encuentros entre equipos de los conjuntos $\{e_i..e_m\}$ y $\{e_{m+1}..e_j\}$, organizándolo de la siguiente manera:

El día $n/2$:

$$\begin{array}{ccccccc} e_i & & e_{i+1} & & \dots & & e_m \\ | & & | & & & & | \\ e_{m+1} & & e_{m+2} & & \dots & & e_j \end{array}$$

El día $n/2 + 1$:

$$\begin{array}{ccccccc} e_i & & e_{i+1} & & \dots & & e_{m-1} & & e_m \\ | & & | & & & & | & & | \\ e_{m+2} & & e_{m+3} & & \dots & & e_j & & e_{m+1} \end{array}$$

Y así sucesivamente hasta el día $n - 1$:

$$\begin{array}{ccccccc} e_i & & e_{i+1} & & \dots & & e_m \\ | & & | & & & & | \\ e_j & & e_{m+1} & & \dots & & e_{j-1} \end{array}$$

Para realizar las rotaciones se utiliza la aritmética modular. El algoritmo que queda para realizar las rotaciones y asignar los equipos y fechas de juego en la tabla T es el siguiente:

fun Combinar(i,j,d:natural)

var

m,n,s,t: natural

a,b: natural

fvar

$m \leftarrow (i+j-1)/2$

```

n ← j-i+1
para s ← 0 hasta n-1 hacer
  para t ← 0 hasta n-1 hacer
    a ← i+t
    b ← (m+s+1) mod (j+1)
    T[a,b] ← d+s
  fpara
fpara
ffun

```

El bucle en s representa las rotaciones. El bucle interior en t representa los partidos para la rotación s . El coste del algoritmo de combinación es cuadrático, considerando $n = j - i + 1$ el tamaño del problema. En tal caso, la ecuación de recurrencia queda $T(n) = 2T(n/2) + cn^2$ con $a = 2$, $b = 2$ y $k = 2$ con lo que siendo $a < b^k$ la ecuación tiene solución cuadrática con coste $O(n^2)$.

4.7 Skyline de una ciudad

Sobre una ciudad se alzan los edificios dibujando una linea de horizonte (conocida como *skyline*). Supongamos una ciudad representada por un conjunto de edificios $C = \{e_1, e_2, \dots, e_n\}$ y cada edificio representado por un rectángulo sobre un eje de coordenadas. El problema consiste en calcular la linea de horizonte de la ciudad, en forma de una secuencia de puntos sobre el plano.

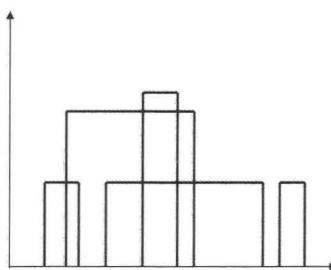


Figura 4.5: Una ciudad, con edificios

El problema se puede descomponer en subproblemas de menor tamaño y de la misma naturaleza. La figura 4.5 ilustra un problema con 5 edificios y su solución se ilustra en la figura 4.6. Supongamos sin perder generalidad que $n = 2^k$ y que elegimos dividir el problema en subproblemas de tamaño $n/2$. La solución de cada subproblema es una linea de horizonte, formada por una lista de puntos en el plano.

Cada subproblema nos proporciona una línea de horizonte, representada por una lista de puntos. La combinación de los subproblemas es recorrer ambas, y elegir en cada momento el punto de mayor ordenada.

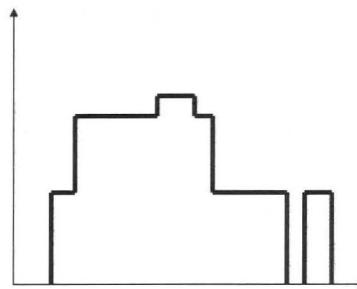


Figura 4.6: *El skyline de la ciudad anterior*

Sea por tanto el algoritmo a partir del esquema general el siguiente:

```
fun DyV(problema)
  si trivial(problema) entonces
    dev solución-trivial
  sino hacer
    { $p_1, p_2, \dots, p_k$ }  $\leftarrow$  descomponer(problema)
    para  $i \in (1..k)$  hacer
       $s_i \leftarrow$  DyV( $p_i$ )
    fpara
    dev combinar( $s_1, s_2, \dots, s_k$ )
ffun
```

Y refinamos el esquema:

Caso trivial: el caso trivial es realizar el skyline de un edificio.

Descomponer: el conjunto de edificios se divide en dos mitades iguales.

Combinar: suponemos, por inducción resuelto el problema, es decir, que la entrada a este algoritmo son dos soluciones consistentes en sendas líneas de horizonte. La combinación consiste en fusionar estas líneas de horizonte eligiendo la mayor ordenada para cada abcisa donde haya edificios.

Nos ocupamos ahora de la estructura de datos. Para representar los edificios podemos elegir tríadas del tipo $e_1 = (x_1, x_2, h)$ con las coordenadas representando la posición inicial y final y la h la altura del mismo. Por otro lado, el tipo de datos *TipoSkyline* que

describe las líneas de horizonte se pueden representar como una concatenación de posiciones y la altura a partir de dicha posición $s = (x_1, h_1, x_2, h_2, \dots, x_k, h_k)$, siendo x la abscisa y h la ordenada (altura) del edificio en ese punto. Cada uno de los pares x_i, h_i representa transiciones bien entre un edificio y otro, o bien entre un edificio y la linea de horizonte. Pasamos a escribir el algoritmo principal a partir de las consideraciones y el refinamiento realizados:

```
fun Edificios(C: vector [1..n] de edificio; i,j:natural): TipoSkyline
  var
    m,n: natural
  fvar
    m  $\leftarrow$  (i+j-1)/2
    n  $\leftarrow$  j-i+1
  si n = 1 entonces
    dev convierte_edificio_en_skyline(C[i])
  sino
    s1  $\leftarrow$  Edificios(C,i,m)
    s2  $\leftarrow$  Edificios(C,m+1,j)
    s  $\leftarrow$  Combinar(s1,s2)
    dev s
  fsi
ffun
```

Nos queda ahora programar las funciones auxiliares. Se deja al lector la función para el caso trivial y nos centramos en la función de combinación, que fusiona dos líneas de horizonte. En este caso, lo apropiado es ir ordenada por ordenada, eligiendo en cada caso, la altura mayor. La idea es llevar dos contadores, uno por cada línea de horizonte.

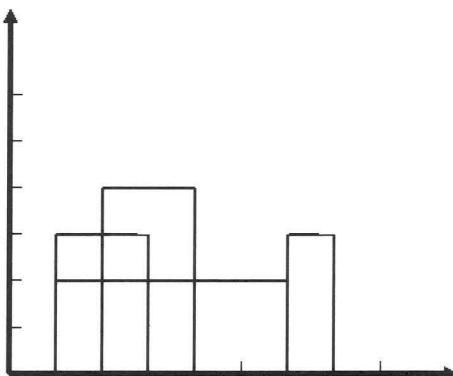


Figura 4.7: Una ciudad, con 4 edificios

```

fun Combinar(s1,s2: TipoSkyline)
  var
    i,j,k: natural
  fvar
    n ← s1.longitud()
    m ← s2.longitud()
     $s1_x \leftarrow$  ExtraeOrdenadas(s1)
     $s1_h \leftarrow$  ExtraeAlturas(s1)
     $s2_x \leftarrow$  ExtraeOrdenadas(s2)
     $s2_h \leftarrow$  ExtraeAlturas(s2)
    i ← 1; j ← 1
    k ← 1; S ← [ ]
  mientras ( $i \leq n$ )  $\vee$  ( $j \leq m$ ) hacer
    x ← min( $s1_x[i], s2_x[j]$ )
    si  $s1_x[i] \leq s2_x[j]$  entonces
      max ← max( $s1_h[i], s2_h[j - 1]$ )
      i ← i+1
    sino
      si  $s1_x[i] > s2_x[j]$  entonces
        max ← max( $s1_h[i - 1], s2_h[j]$ )
        j ← j+1
      sino
        max ← max( $s1_h[i], s2_h[j]$ )
        i ← i+1
        j ← j+1
    fsi
  fsi
     $S_x[k] \leftarrow x$ 
     $S_h[k] \leftarrow max$ 
    k ← k+1
  fmientras
  S ←  $S_x \cup S_h$ 
  dev S
ffun

```

Finalmente apliquemos el algoritmo al ejemplo de los 4 edificios de la figura 4.7, representados por el vector:

$$C_1 = \{(1,3,3)(1,6,2)(2,4,4)(6,7,3)\}$$

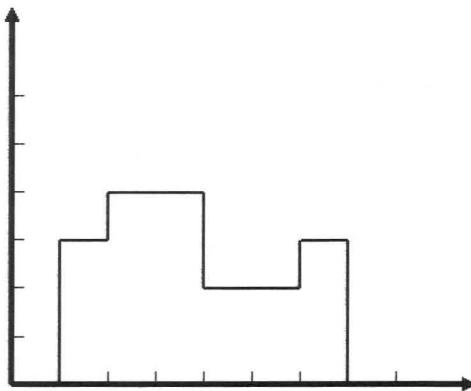


Figura 4.8: Línea de horizonte resultante

El resultado sería la secuencia siguiente, correspondiente con la figura 4.8:

$$S = (1, 3, 2, 4, 4, 2, 6, 3, 7, 0)$$

4.8 Ejercicios propuestos

1. Se pide realizar la multiplicación de dos polinomios, representados mediante dos vectores de coeficientes, usando la técnica de divide y vencerás. Obtener una complejidad mejor que $O(n^3)$.
2. Se tienen 3 postes y n discos horadados e insertables en los postes. Los discos son todos de diferentes diámetros, y se disponen inicialmente en el primer poste ascendentemente de mayor a menor diámetro. El problema consiste en pasar los discos del poste 1 al 3 con las siguientes condiciones: (1) solo se puede mover un disco al mismo tiempo y (2) no se puede colocar un disco sobre otros de menor diámetro.
3. Considerando que disponemos de una representación en forma de vector de números grandes, representados los dígitos por valores del vector, se pide diseñar un algoritmo que proporcione un mecanismo de multiplicación de dos números.
4. Se tiene un vector de enteros no repetidos y ordenados de menor a mayor. Diseñar un algoritmo que compruebe en tiempo logarítmico si existe algún elemento del vector que coincida con su índice.
5. Implementar una función $\text{exponencial}(x, n)$ que calcule x^n usando la estrategia de *divide y vencerás*.

6. Diseñar un algoritmo que permita el cálculo de la expresión $f_n = af_{n-3} + bf_{n-2} + cf_{n-1}$ en tiempo logarítmico. Utilizar para ello la expresión:

$$\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ a & b & c \end{pmatrix} \begin{pmatrix} f_{n-3} \\ f_{n-2} \\ f_{n-1} \end{pmatrix} = \begin{pmatrix} f_{n-2} \\ f_{n-1} \\ f_n \end{pmatrix}$$

Parametrizar la función para calcular el término n -ésimo de la función de *fibonacci*.

7. Dado un vector de naturales con algunos elementos duplicados, eliminar los elementos duplicados del vector mediante un algoritmo con coste $O(n \log n)$
8. Resolver el elemento mayoritario de un vector A en tiempo lineal mediante una estrategia de divide y vencerás. Para ello empareja los elementos de A arbitrariamente para obtener $n = 2$ pares. Para cada par, si ambos son diferentes, descartarlos, si son iguales, dejar uno. Comprobar que tras este paso, podemos dejar de considerar al menos $n = 2$ elementos de A y que el resto del vector tiene un elemento mayoritario si y solo si A lo tiene.
9. Dada una nube N de k puntos en un plano $N = \{(a_1, b_1), (a_2, b_2), \dots, (a_k, b_k)\}$, hallar los dos puntos más cercanos entre ellos mediante un algoritmo de tipo divide y vencerás.

4.9 Notas bibliográficas

El esquema de divide y vencerás está descrito ampliamente en [BB06] y algunos de los ejercicios en [GA97]. Por otro lado, en la parte de ordenación, la referencia clásica es [Knu73] y [BB06] y la elaboración didáctica está basada en parte en [GA01].

Algunos de los ejemplos mencionados han sido sacados de textos clásicos como [AP89], [AHU98] y [BB06]. Varios de los ejercicios propuestos, como el de la sucesión de *fibonacci* generalizada y el del elemento que coincide con su índice están resueltos en [GA97].

Capítulo 5

Programación dinámica

Este capítulo está dedicado a presentar la técnica de la Programación Dinámica. Esta técnica se caracteriza por registrar resultados parciales que se producen durante la resolución de algunos problemas y que se utilizan repetidamente. De esta forma se consigue reducir el coste de cómputo, al evitar la repetición de ciertos cálculos.

Presentamos el esquema general ejemplificando su aplicación al problema de la construcción de la sucesión de los número de Fibonacci. Después se presentan diversos problemas en los que esta técnica nos permite reducir el coste de cómputo: los coeficientes binomiales, la devolución de cambio, el viaje por el río, la mochila, la multiplicación asociativa de matrices, la búsqueda del camino de coste mínimo entre nodos de un grafo dirigido, y la distancia de edición. En cada caso se estudian la forma que toman los distintos elementos del esquema para el problema tratado y el coste del algoritmo resultante. El estudio detenido de estos problemas debe permitir al lector abordar los problemas que se proponen al final del capítulo. Para resolverlos se seguirán los pasos indicados en los problemas desarrollados a lo largo del capítulo. Antes de abordar este tema se aconseja haber estudiado el capítulo dedicado a la técnica de *Divide y Vencerás*, que en algunos problemas comparte ciertos elementos con el esquema que ahora se presenta.

5.1 Planteamiento general

La programación dinámica es una técnica mediante la que se reduce el coste de ejecución de un algoritmo memorizando soluciones parciales que se necesitan para llegar a la solución final. Algunos casos en los que es aplicable son problemas que también se pueden abordar con el esquema “divide y vencerás”, presentado en el capítulo 4. En este esquema un problema se va dividiendo en subproblemas de tamaño menor que el original. Estas divisiones sucesivas del problema se plantean generalmente en forma de algoritmos recursivos. Una condición para que estos algoritmos sean eficientes es que no haya llamadas repetidas en la secuencia de llamadas recursivas. Cuando ocurren estas repeticiones conviene recurrir al esquema de programación dinámica, almacenando

los resultados ya calculados para reutilizarlos. La programación dinámica también es aplicable a muchos problemas de optimización cuando se cumple el *Principio de Optimalidad de Bellman* [Bel57] que dice que dada una secuencia óptima de decisiones, toda subsecuencia de ella es, a su vez, óptima. La resolución del problema requerirá buscar la solución óptima para muchas subsecuencias que pudieran formar parte de la secuencia óptima final.

Generalmente los resultados parciales que se producen al aplicar algoritmos correspondientes a este esquema se almacenan en una tabla, de la que se toman cuando el algoritmo los vuelve a necesitar. Los primeros datos que se almacenan corresponden a los subproblemas más sencillos, y a partir de ellos se van construyendo de forma incremental las soluciones a subproblemas mayores, hasta llegar a la del problema completo. La reducción del coste en tiempo de ésta técnica supone un incremento en el coste espacial, ya que el almacenamiento de las soluciones parciales requiere incrementar el espacio dedicado a los datos.

La forma de aplicación es muy dependiente del problema y de las estructura de datos que se utilicen en su resolución. Sin embargo, podemos establecer la siguiente secuencia de acciones que se requieren para la aplicación de esta técnica:

- Establecimiento de las ecuaciones que representan el problema.
- Identificación de los resultados parciales.
- Construcción de la tabla de resultados parciales:
 - Inicialización de la tabla con los casos base que establece la ecuación del problema.
 - Establecimiento del orden de llenado de la tabla, de forma que se calculen en primer lugar los resultados parciales que requieren pasos posteriores.
 - Sustitución de las llamadas recursivas del algoritmo por consultas a la tabla.

Para entender el funcionamiento del esquema veamos como se aplican estos puntos a la resolución de un problema muy sencillo, la construcción de la sucesión de los números de Fibonacci. Esta sucesión se puede representar por la ecuación:

$$\text{Fibonacci}(n) = \begin{cases} 1 & \text{si } n = 0, 1 \\ \text{Fibonacci}(n - 1) + \text{Finonacci}(n - 2) & \text{si } n > 1 \end{cases}$$

que se implementa de forma directa por el siguiente algoritmo recursivo:

```
fun Fib(n: entero): entero
  si n ≤ 1 entonces
    dev 1
  sino
```

dev Fib(n-1) + Fib(n-2)

fsi

ffun

Sin embargo, este algoritmo tiene un tiempo de ejecución que crece exponencialmente con n . Podemos mejorar la eficiencia diseñando un algoritmo de coste lineal si vamos almacenando los resultados parciales de la serie. Por ejemplo, para calcular $Fibonacci(10)$ necesitamos conocer el valor de $Fibonacci(9)$ y de $Fibonacci(8)$, pero $Fibonacci(8)$ ya lo hemos calculado al calcular $Fibonacci(9)$. Lo mismo ocurre con términos anteriores de la serie. Por lo tanto basta utilizar una tabla para almacenar los términos nuevos que se van calculando.

| | | | |
|--------------|--------------|-----|--------------|
| Fibonacci(0) | Fibonacci(1) | ... | Fibonacci(n) |
|--------------|--------------|-----|--------------|

Ahora podemos escribir un algoritmo iterativo de coste lineal para calcular la secuencia utilizando los valores almacenados en la tabla:

fun FibDin(n : entero): entero

var

i,suma: entero

t: tabla[0..1] de entero

fvar

si $n \leq 1$ **entonces**

dev 1

sino

$t[0] \leftarrow 1$

$t[1] \leftarrow 1$

para $i \leftarrow 2$ **hasta** n **hacer**

$t[i] \leftarrow t[i-1] + t[i-2]$

fpara

fsi

ffun

Podemos reducir la complejidad espacial del algoritmo si notamos que en este caso sólo necesitamos los dos últimos términos almacenados. Si almacenamos únicamente los dos últimos términos podemos prescindir de la tabla y la complejidad espacial pasa de $O(n)$ a $O(1)$ con el siguiente algoritmo:

```

fun FibDin2(n: entero): entero
    var
        i,suma,f,g: entero
    fvar
        si n ≤ 1 entonces
            dev 1
        sino
            f ← 1
            g ← 1
            para i ← 2 hasta n hacer
                suma ← f + g
                g ← f
                f ← suma
            fpara
        fsi
ffun

```

A continuación vamos a ver varios problemas en los que la programación dinámica nos permite diseñar algoritmos eficientes. Tendremos así una perspectiva de los casos más comunes de aplicación y sus detalles técnicos.

5.2 Los coeficientes binomiales

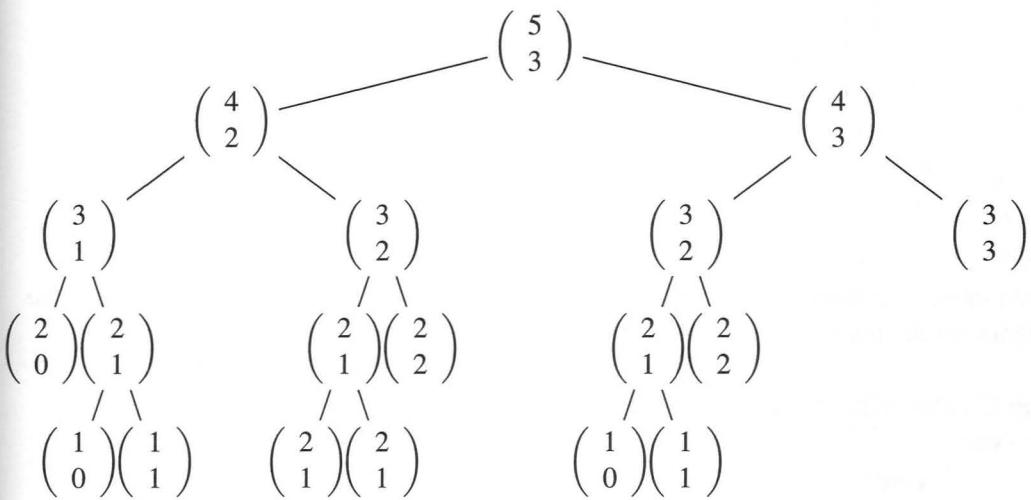
Los coeficientes binomiales se utilizan en combinatoria para calcular combinaciones de elementos, es decir, el número de formas en que se pueden extraer distintos subconjuntos a partir de un conjunto dado, sin importar su orden. El siguiente ejemplo que vamos a considerar es el cálculo de los coeficientes binomiales, definidos de la siguiente forma:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

que alternativamente se puede escribir como:

$$\binom{n}{k} = \begin{cases} 1 & \text{si } k = 0 \text{ ó } k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{si } 0 < k < n \end{cases}$$

El algoritmo que implementa directamente la ecuación recursiva anterior tiene complejidad exponencial, ya que repite numerosos cálculos. Por ejemplo, para calcular $\binom{5}{3}$ tendríamos las llamadas que aparecen en la siguiente figura:



Podemos ver que incluso para un número combinatorio tan pequeño como el de este ejemplo hay muchas llamadas que se repiten.

Conseguiremos un algoritmo más eficiente almacenando los coeficientes que se van calculando. Estos coeficientes, calculados de forma ascendente, constituyen el famoso *Triángulo de Pascal*:

$$\begin{array}{cccc}
 & & C(0, 0) & \\
 & C(0, 0) & C(1, 0) & \\
 C(2, 0) & C(2, 1) & C(2, 2) & \\
 C(3, 0) & C(3, 1) & C(3, 2) & C(3, 3) \\
 \dots & & &
 \end{array}$$

En esta estructura cada coeficiente se obtiene sumando los dos elementos superiores a él, y la complejidad temporal para calcular $C(n, k)$ es de orden $O(nk)$. Podemos almacenar estos coeficientes en una tabla de la siguiente forma:

| | 0 | 1 | 2 | 3 | ... | $k - 1$ | k |
|-------|-----|-----|-----|-----|--------------|---------|-------------------------|
| 0 | 1 | | | | | | |
| 1 | 1 | 1 | | | | | |
| 2 | 1 | 2 | 1 | | | | |
| 3 | 1 | 3 | 3 | 1 | | | |
| ... | ... | ... | ... | ... | ... | | |
| $n-1$ | ... | ... | ... | ... | $C(n-1,k-1)$ | | $C(n-1,k)$ |
| n | ... | ... | ... | ... | ... | | $C(n-1,k-1) + C(n-1,k)$ |

Esta tabla se construye de arriba abajo y de izquierda a derecha mediante el siguiente algoritmo iterativo:

```

fun CoefBin( $n, k$ : entero): entero
  var
    i,j: entero
    t: Tabla[0..n] de entero
  fvar
  si  $k \leq 0 \vee k = n$  entonces
    dev 1
  sino
    para i  $\leftarrow 0$  hasta n hacer t[i,0]  $\leftarrow 1$  fpara
    para i  $\leftarrow 1$  hasta n hacer t[i,1]  $\leftarrow i$  fpara
    para i  $\leftarrow 2$  hasta k hacer t[i,i]  $\leftarrow 1$  fpara
    para i  $\leftarrow 3$  hasta n hacer
      para j  $\leftarrow 2$  hasta n-1 hacer
        si j  $\leq k$  entonces
          t[i,j]  $\leftarrow t[i-1,j-1] + t[i-1,j]$ 
        fsi
      fpara
    fpara
  fsi
ffun

```

5.3 Devolución de cambio

Recordemos la formulación del problema de la devolución de cambio planteado en el capítulo 3, dedicado a los algoritmos voraces. Se tiene un conjunto de N tipos de monedas, cada una con un valor x_i . Se supone que contamos con una cantidad ilimitada de monedas de cada tipo. El problema consiste en hallar el número mínimo de monedas que necesitamos para dar una cierta cantidad C .

Un algoritmo voraz que eligiese siempre la moneda de mayor valor que se pueda tomar para acercarse a la cantidad C no funcionaría para cualquier conjunto de tipos de monedas. Por ejemplo, si tenemos monedas de valores 1, 6 y 10, y la cantidad a completar fuera 24, la estrategia voraz elegiría las siguientes monedas: 10, 10, 1, 1, 1 y 1. Es decir se darían 6 monedas, cuando es posible devolver esa cantidad con sólo 4 monedas de valor 6.

Veamos como resolver este problema con el esquema de programación dinámica. En primer lugar debemos pensar como plantear el problema de forma incremental. Consideramos el tipo de moneda de mayor valor, x_N . Si $x_N > C$ entonces la descartamos y pasamos a considerar monedas de menor valor. Si $x_N \leq C$ tenemos dos opciones: o tomar una moneda de tipo x_N , y completar la cantidad restante $C - x_N$ con otras monedas, o no tomar ninguna moneda de tipo x_N y completar la cantidad C con monedas de menor valor. De las dos opciones nos quedamos con la que requiera un número menor de monedas. El problema lo podemos expresar de la siguiente forma cuando consideramos N tipos de monedas:

$$\text{cambio}(N, C) = \begin{cases} \text{cambio}(N-1, C) & \text{si } x_N > C \\ \min\{\text{cambio}(N-1, C), \text{cambio}(N, C - x_N) + 1\} & \text{si } x_N \leq C \end{cases}$$

Podemos razonar análogamente para monedas de valores k menores que N y para cantidades C' menores que C :

$$\text{cambio}(k, C') = \begin{cases} \text{cambio}(k-1, C') & \text{si } x_k > C' \\ \min\{\text{cambio}(k-1, C'), \text{cambio}(k, C' - x_k) + 1\} & \text{si } x_k \leq C' \end{cases}$$

Llegamos a los casos base de la recurrencia cuando completamos la cantidad C :

$$\text{cambio}(k, 0) = 0 \text{ si } 0 \leq k \leq n,$$

o cuando ya no quedan más tipos de monedas por considerar, pero aún no se ha completado la cantidad C :

$$\text{cambio}(0, C') = \infty \text{ si } 0 < C' \leq C.$$

Podemos construir una tabla para almacenar los resultados parciales que tenga una fila para cada tipo de moneda y una columna para cada cantidad posible entre 1 y C . Cada posición $t[i, j]$ será el número mínimo de monedas necesario para dar una cantidad j , con $0 \leq j \leq C$, utilizando sólo monedas de los tipos entre 1 e i , con $0 \leq i \leq N$. La solución al problema será por tanto el contenido de la casilla $t[N, C]$. Para construir la tabla empezamos llenando los casos base $t[i, 0] = 0$, para todo i con $0 \leq i \leq N$. A continuación podemos llenar la tabla bien por filas de izquierda a derecha, o bien por columnas de arriba a abajo.

Consideremos un ejemplo en el que se quiere pagar una cantidad de 12 unidades utilizando monedas de valores 1, 6 y 10. La siguiente tabla muestra la situación cuando llenaremos por columnas, después de la inicialización, que pone a 0 toda la primera columna, y de llenar las siguientes 5 columnas:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|------------|---|---|---|---|---|---|---|---|---|---|----|----|----|
| $x_1 = 1$ | 0 | 1 | 2 | 3 | 4 | 5 | | | | | | | |
| $x_1 = 6$ | 0 | 1 | 2 | 3 | 4 | 5 | | | | | | | |
| $x_1 = 10$ | 0 | 1 | 2 | 3 | 4 | 5 | | | | | | | |

La casilla $t[x_1 = 1,1]$ contiene el número de monedas de valor 1 necesarias para completar la cantidad 1, es decir una. La casilla $t[x_1 = 6,1]$ contiene un número de monedas de valor 6 o menor que 6 (en nuestro ejemplo 1) necesarias para dar la cantidad 1, que de nuevo es una. Y finalmente $t[x_1 = 10,1]$ es la cantidad de monedas de valor 10, 6 o 1 necesarias para completar 1, que también es una moneda. El mismo razonamiento se aplica para las columnas dos a cinco. Cuando llegamos a la columna 6, la casilla $t[x_1 = 1,6]$ toma el valor 6, ya que se necesitan seis monedas de valor 1 para llegar a 6. Pero en la siguiente casilla de la columna, $t[x_1 = 6,6]$ tenemos dos alternativas para conseguir el valor 6 con monedas de valores 1 y 6: o seis monedas de valor 1 o una moneda de valor 6. Como la última alternativa da un número menor de monedas, el valor que se incluye en la casilla es 1. El mismo razonamiento se sigue para la última casilla de la columna, y también para las siguientes columnas hasta la 9, como muestra la siguiente tabla:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|------------|---|---|---|---|---|---|---|---|---|---|----|----|----|
| $x_1 = 1$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | | | |
| $x_1 = 6$ | 0 | 1 | 2 | 3 | 4 | 5 | 1 | 2 | 3 | 4 | | | |
| $x_1 = 10$ | 0 | 1 | 2 | 3 | 4 | 5 | 1 | 2 | 3 | 4 | | | |

Llegamos ahora a la columna 10. Siguiendo los razonamientos anteriores necesitamos diez monedas de valor 1 para completar la cantidad. Si consideramos monedas de valores 1 y 6 necesitamos un mínimo de cinco, una de 6 y cuatro de 1. Y si consideramos monedas de valores 1, 6 y 10, lo que ocurre en la última casilla de la columna, necesitamos un mínimo de una moneda. El razonamiento es análogo para la columna 11. Llegamos a la columna correspondiente a la cantidad 12. En la primera casilla indicamos que se necesitan 12 monedas de valor 1. Pero al llegar a la siguiente casilla, debemos darnos cuenta de que ahora tenemos tres alternativas posibles de completar 12 con monedas de valores 1 y 6: podemos tomar doce monedas de valor 1, una de valor 6 y seis de valor 1, lo que hace un total de siete, o dos monedas de 6. Nos quedamos con la última alternativa que es la mejor. Para la última casilla tenemos aún otra posibilidad más, que es una moneda de 10 y dos de 1, lo que hace un total de tres monedas. Pero como esta alternativa es peor que la anotada en la casilla anterior, nos quedamos con las dos monedas de 6, que es la respuesta final al problema. La siguiente tabla muestra la situación final:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|------------|---|---|---|---|---|---|---|---|---|---|----|----|----|
| $x_1 = 1$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| $x_1 = 6$ | 0 | 1 | 2 | 3 | 4 | 5 | 1 | 2 | 3 | 4 | 5 | 6 | 2 |
| $x_1 = 10$ | 0 | 1 | 2 | 3 | 4 | 5 | 1 | 2 | 3 | 4 | 1 | 2 | 2 |

El algoritmo para el problema puede tomar la siguiente forma:

```

tipo Tabla = matriz[1..N,1..C] de entero
tipo Vector = matriz[0..N] de entero
fun DarCambio(C: entero, moneda: Vector): Tabla
    var
        t: Tabla
        i,j: entero
    fvar
    para i ← 1 hasta N hacer
        t[i,0] ← 0
    fpara
    para j ← 1 hasta C hacer
        para i ← 1 hasta N hacer
            si i = 1 ∧ moneda[i] > j entonces
                t[i,j] ← ∞
            sino
                si i = 1 entonces
                    t[i,j] ← 1 + t[1,j-moneda[i]]
                sino
                    si j < moneda[i] entonces
                        t[i,j] ← t[i-1,j]
                    sino
                        t[i,j] ← min(t[i-1,j], t[i,j-moneda[i]] + 1)
                    fsi
                fsi
            fpara
            dev t
    ffun

```

El algoritmo comienza inicializando la primera columna de la tabla a 0. Después va completando las restantes columnas hasta llegar a la C . Para cada columna empieza por comprobar si el valor de la moneda más pequeña es mayor que la cantidad a completar. Si es así se coloca una marca en la tabla ∞ para indicar la imposibilidad de hacer la operación. Si no es así, se asigna la casilla correspondiente a la moneda más pequeña. Si no estamos en la primera de la filas se comprueba si la cantidad a completar j es menor que la moneda asociada a esa fila. Si no es así no podemos utilizar esa moneda y por tanto, el valor de esa casilla no cambia respecto del de la fila anterior. Si es menor, entonces

podemos encontrar nuevas formas de completar la cantidad j , de las que seleccionamos la mejor.

Para estudiar el coste del algoritmo calculamos el tamaño de la tabla que hay que construir: $N \times (C+1)$. Como las operaciones aritméticas involucradas son de coste constante, el tiempo de ejecución está en $\Theta(NC)$.

Si lo que queremos es que el algoritmo no sólo nos diga cuál es la cantidad mínima de monedas, sino también qué monedas hay que devolver tenemos dos alternativas. Una de ellas es ir almacenando en la tabla no sólo el número de monedas, sino también los tipos de las monedas. Esto llevaría a un aumento importante del coste de memoria. La mejor alternativa es utilizar la tabla construida para saber el número de monedas. Partimos de la casilla final. Para cada casilla $t[i,j]$ el algoritmo va comprobando si su valor ha variado respecto a la casilla de la fila superior. Si no ha variado podemos deducir que no se ha empleado ninguna moneda del tipo de la fila i , y pasamos a comprobar la casilla superior $t[i-1,j]$. Si ha variado, anotamos que se ha utilizado una moneda de ese tipo x_i y nos movemos a la casilla $t[i, j-\text{moneda}[i]]$, para ver qué monedas se han utilizado para dar la cantidad restante. Siguiendo esta estrategia terminaremos alcanzando la casilla $t[0,0]$ en la que ya no queda ninguna cantidad pendiente. El algoritmo puede tomar la forma que se muestra a continuación:

```

tipo Tabla = matriz[1..N,1..C] de entero
tipo Vector = matriz[0..N] de entero
fun seleccionar_monedas(C: entero, moneda: Vector, t:Tabla, seleccion: Vector)
    var
        i,j: entero
    fvar
    para i  $\leftarrow$  0 hasta N hacer
        seleccion[i]  $\leftarrow$  0
    fpara
        i  $\leftarrow$  N
        j  $\leftarrow$  C
    mientras j  $>$  0 hacer
        si i  $>$  1  $\wedge$  t[i,j] = t[i-1,j] entonces
            i  $\leftarrow$  i - 1
        sino
            seleccion[i]  $\leftarrow$  seleccion[i] + 1
            j  $\leftarrow$  j - moneda[i]
        fsi
    fmientras
ffun

```

Se trata de un algoritmo voraz cuyo coste está en $\Theta(C)$, que será el número de casillas que recorrerá como máximo. El vector *selección* almacena en la casilla i el número de monedas de tipo i utilizadas.

5.4 El viaje por el río

A lo largo de un río hay N embarcaderos. En cualquiera de ellos se puede alquilar una barca para ir a cualquier otro embarcadero que esté río abajo, ya que el río presenta una corriente muy fuerte y no es posible navegar río arriba. Las tarifas de alquiler de las barcas, que son distintas para cada par de embarcaderos, pueden consultarse en una tabla T . Para cualquier par de embarcaderos i y j tales que $i < j$, a la posición $T[i, j]$ de esta tabla se le asigna el coste T_{ij} de ir del embarcadero i al j , representado en la figura 5.1. A veces ocurre que el alquiler entre dos embarcados i y j es más costoso que el alquiler de barcas para una sucesión de viajes más cortos a embarcaderos intermedios entre i y j , teniendo en cuenta que el cambio de barca no supone coste adicional.

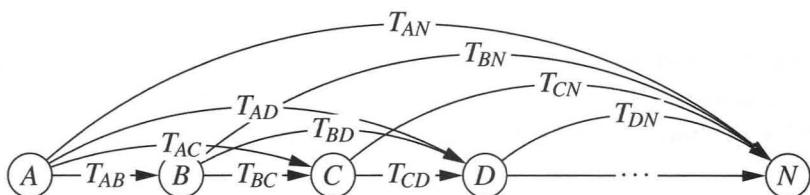


Figura 5.1: Ejemplo de una serie de embarcaderos A, B, \dots, N por el río. Los arcos indican los costes de los distintos trayectos.

Buscamos un algoritmo eficiente para encontrar la forma más barata de ir de uno de los embarcaderos a cualquier otro, que esté río abajo.

Se trata de un problema de optimización en el que se cumple el principio de optimalidad. Si el coste de ir del embarcadero e_i a otro embarcadero e_j pasando por un embarcadero intermedio e_k ,

$$C(e_i, e_j) = T[e_i, e_k] + C(e_k, e_j)$$

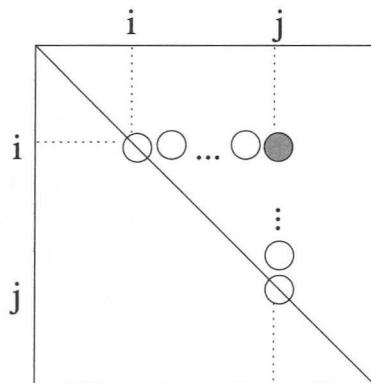
es óptimo, entonces el coste $C(e_k, e_j)$ de ir de ese embarcadero intermedio al final del trayecto también tiene que ser óptimo.

Podemos plantear las ecuaciones de recurrencia para el problema de la siguiente forma:

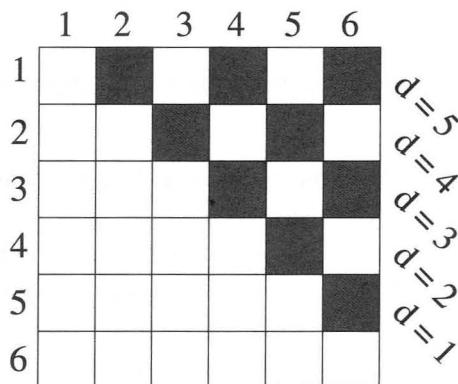
$$C(i, j) = \begin{cases} 0 & \text{si } i = j \\ \min_{i < k \leq j} \{T[i, k] + C(k, j)\} & \text{si } i < j \end{cases}$$

La función *min* calcula el mínimo del coste de todas las combinaciones posibles de trayectos haciendo una escala en algún embarcadero intermedio k , incluyendo el caso en que $k = j$, es decir, que el trayecto más barato puede ser también el viaje directo.

Nuestro siguiente paso es construir la estructura que va a almacenar las soluciones parciales. En este caso, se trata de una matriz C de tamaño $N \times N$ siendo N el número de embarcaderos. De esta matriz sólo necesitamos la mitad por encima de la diagonal principal. La siguiente figura muestra las casillas que necesitamos para calcular una nueva casilla $T[i, j]$, que son las casillas de la fila i entre la diagonal y la columna j , y las casillas de la columna j entre la diagonal en la fila i :



Una forma de construir la tabla de resultados parciales es por diagonales de la mitad superior de la matriz. Las diagonales se van completando siguiendo el orden dado por la numeración que muestra la siguiente figura para el caso $N = 6$:



Llamamos diagonal 1 ($d = 1$, en negro) a la que está por encima de la diagonal principal y que contiene $N - 1$ elementos. La diagonal 2 (en blanco) es la siguiente por encima de ella, y así sucesivamente hasta llegar a la diagonal $N - 1$ que contiene un único elemento. La tabla se inicializa asignando 0 a los valores de la diagonal principal.

El algoritmo para resolver este problema puede tomar la siguiente forma, siendo N el número de embarcaderos del problema:

tipo Tabla = matriz[1..N,1..N] de entero
fun ViajeRio(T: Tabla, N: entero, C: Tabla)

```

var
    i,diag: entero
fvar
para i  $\leftarrow$  1 hasta N hacer
    C[i,i]  $\leftarrow$  0
fpara
para diag  $\leftarrow$  1 hasta N-1 hacer
    para i  $\leftarrow$  1 hasta N-diag hacer
        C[i,i+diag]  $\leftarrow$  MinMultiple(C,i,i+diag)
    fpara
fpara
ffun
```

Los datos de entrada del algoritmo son la matriz de precios de trayectos T y el número de embarcaderos N . La salida es la matriz de costes mínimos de trayecto C . Tras inicializar la diagonal principal, el algoritmo pasa a completar secuencialmente las diagonales de la 1 a la $N - 1$. Para la diagonal 1 ($diag = 1$) se rellenan las casillas $C[1,2], C[2,3], \dots, C[N - 1,N]$. Para la siguiente diagonal, 2, se rellenan las casillas $C[1,3], C[2,4], \dots, C[N - 2,N]$. Y así sucesivamente, teniendo al final el coste mínimo del trayecto entre cualquier par de embarcaderos.

La función que calcula el mínimo entre múltiples valores, *MinMultiple* puede tomar la siguiente forma:

fun MinMultiple(C: Tabla, i: entero, j: entero): entero

```

var
    k, minimo: entero
fvar
    minimo  $\leftarrow$   $\infty$ 
para k  $\leftarrow$  i+1 hasta j hacer
    minimo  $\leftarrow$  min(minimo, C[k,j] + T[i,k])
fpara
dev minimo
ffun
```

donde la función $\min(a,b)$ devuelve el valor mínimo entre a y b .

El coste temporal de este algoritmo está en $O(N^3)$ ya que se tienen dos bucles anidados de tamaño N que realizan una llamada a una función de orden $O(N)$.

Podemos plantear el problema de forma que no sólo nos preguntemos cuál es el coste

mínimo de los trayectos, sino también a qué plan de viaje corresponde ese coste mínimo, es decir, los embarcaderos por los que hay que pasar. Para poder dar esta información necesitamos una tabla adicional $E[1..N, 1..N]$ que registre el embarcadero en el que hemos tenido que hacer escala para alcanzar el coste mínimo. Esta tabla se inicializa a 0 indicando que inicialmente no hay escalas, y después se va modificando a la vez que se rellena la matriz de costes C . Para ello modificamos la función *MinMultiple* para que devuelva también el embarcadero con cuya escala se consigue el mínimo:

```
fun MinMultiple2(C: Tabla, i: entero, j: entero, minimo: entero, emb: entero)
    var
        k, tmp: entero
    var
        minimo ← ∞
        emb ← i
    para k ← i+1 hasta j hacer
        tmp ← minimo
        si C[k,j] + T[i,k] < minimo entonces
            minimo ← C[k,j] + T[i,k]
            emb ← k
        fsi
    fpara
ffun
```

El algoritmo para este caso se modifica para que almacene en la nueva tabla E de embarcaderos (escalas) el otro valor devuelto por la función *MinMultiple2*:

```
tipo Tabla = matriz[1..N,1..N] de entero
fun ViajeRio2(T: Tabla, N: entero, var C: Tabla, var E: Tabla)
    var
        i,diag,min,emb: entero
    var
        para i ← 1 hasta N hacer
            C[i,i] ← 0
        fpara
        para diag ← 1 hasta N-1 hacer
            para i ← 1 hasta N-diag hacer
                MinMultiple2(C,i,i+diag,min,emb)
                C[i,i+diag] ← min
                E[i,i+diag] ← emb
            fpara
        fpara
ffun
```

5.5 La mochila

El problema de la mochila se puede plantear en una versión diferente a la que se vio en el capítulo dedicado a los algoritmos voraces. Se tiene una mochila con una capacidad máxima V y n objetos con volúmenes $v = (v_1, v_2, \dots, v_n)$ y beneficios $b = (b_1, b_2, \dots, b_n)$. Los valores de los volúmenes son enteros. El objetivo de este problema es encontrar una selección de objetos cuya suma de volúmenes no supere la capacidad máxima de la mochila, y de forma que la suma de beneficios sea máxima. Estamos pues ante un problema de optimización con restricciones que podemos plantear de la siguiente forma:

$$\text{maximizar } \sum_{i=0}^n x_i b_i \text{ cumpliendo } \sum_{i=0}^n x_i v_i \leq V$$

donde x_i toma el valor 0 ó 1, 0 para indicar que el objeto i no se incluye en la mochila y 1 para indicar lo contrario.

Considerábamos en la versión resuelta con el algoritmo voraz que los objetos podían partirse, pudiendo incluir en la mochila fracciones de ellos. Ahora consideramos que los objetos son indivisibles, con lo que sólo tenemos la opción de incluirlos o excluirlos. Este hecho cambia completamente la situación, haciendo que un algoritmo voraz ya no sea aplicable. Veamos como podemos plantear este problema utilizando el esquema de programación dinámica.

Formulamos el problema de forma incremental, planteando las ecuaciones recursivas para una función $mochila(i, W)$ que nos da el máximo beneficio para un volumen W que queda libre en la mochila considerando los objetos entre 1 e i , siendo $i \leq n$. Cuando pasamos a considerar el objeto i tenemos dos posibilidades: que el objeto excede la capacidad de la mochila o bien que quepa en ella. En el primer caso se prueba con el resto de los objetos. En el segundo caso tenemos de nuevo dos opciones: o bien lo incluimos, con lo que el beneficio b_i se añade al valor de la función, y el volumen v_i se resta del espacio libre, o bien no lo incluimos, con lo que tenemos que resolver el problema considerando la serie de objetos entre 1 y $i - 1$. Nos quedamos con la opción que maximice el beneficio total. Los casos base del problema se presentan cuando la capacidad de la mochila llega a 0, o cuando no queda ningún objeto. En estos casos el beneficio es 0. También podemos considerar como casos base las configuraciones no válidas a las que se puede llegar cuando se excede la capacidad de la mochila. En este caso asignamos un valor especial “ $-\infty$ ”, que es superado por el beneficio de cualquier configuración válida. Formulamos entonces la ecuación de recurrencia del problema de la siguiente forma:

$$mochila(i, W) = \begin{cases} 0 & \text{si } i = 0 \text{ y } W \geq 0 \\ -\infty & \text{si } W < 0 \\ \max\{mochila(i - 1, W), \\ \quad b_i + mochila(i - 1, W - v_i)\} & \text{si } i > 0 \text{ y } v_i > W \\ \max\{mochila(i - 1, W), \\ \quad b_i + mochila(i - 1, W - v_i)\} & \text{si } i > 0 \text{ y } v_i \leq W \end{cases}$$

En este problema se cumple el principio de optimalidad, ya que si el problema se resuelve de forma óptima también se tienen que haber resuelto de forma óptima los subproblemas en que se ha descompuesto.

Para resolver el problema construimos una tabla $M[n, V]$ con tantas filas como objetos y tantas columnas como indique el volumen V . Sin embargo, esta solución sólo es posible si los volúmenes de los objetos son enteros. Para calcular la posición $M[i, j]$ necesitamos haber calculado dos de las posiciones de la fila anterior. Por tanto construimos la tabla por filas y el valor $M[n, V]$ de la última fila nos da la solución al problema. Aunque, bastaría con almacenar la última fila construida en un vector, si queremos reconstruir el camino para dar información no sólo del máximo beneficio, sino también de con qué objetos se consigue, necesitamos la tabla.

Podemos escribir el algoritmo que resuelve el problema de la siguiente forma:

```

tipo Tabla = matriz[0..n,0..V] de entero
tipo Vector = matriz[0..n] de entero
fun MochilaEntera(vol:Vector, ben:Vector, n: entero, V: entero, M:Tabla)
    var
        i,j: entero
    fvar
    para i  $\leftarrow$  1 hasta n hacer
        M[i,0]  $\leftarrow$  0
    fpara
    para j  $\leftarrow$  1 hasta V hacer
        M[0,j]  $\leftarrow$  0
    fpara
    para i  $\leftarrow$  1 hasta n hacer
        para j  $\leftarrow$  1 hasta V hacer
            si vol[i]  $>$  j entonces
                M[i,j]  $\leftarrow$  M[i-1,j]
            sino
                M[i,j]  $\leftarrow$  max(M[i-1,j], M[i-1,j-vol[i]] + ben[i])
            fsi
        fpara
    fpara
ffun
```

El coste de este algoritmo está en $O(nV)$ por los dos bucles anidados para la construcción de la tabla. El coste espacial, que viene dado por el tamaño de la tabla, está en este mismo orden.

La siguiente tabla muestra un ejemplo del funcionamiento del algoritmo. Se considera un caso con cinco objetos a los que corresponden los valores y beneficios que aparecen

en la tabla, y un volumen máximo de 8. La tabla se rellena de arriba a abajo y de izquierda a derecha. De acuerdo con los resultados de la tabla, vemos que el máximo beneficio que se puede obtener con los objetos introducidos en la mochila es de 19.

| Límite de volumen | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---------------------|---|---|---|---|----|----|----|----|----|
| posición 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $v_1 = 1, b_1 = 2$ | 0 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| $v_2 = 3, b_2 = 5$ | 0 | 2 | 2 | 5 | 7 | 7 | 7 | 7 | 7 |
| $v_3 = 4, b_3 = 10$ | 0 | 2 | 2 | 5 | 10 | 12 | 12 | 15 | 15 |
| $v_4 = 5, b_4 = 14$ | 0 | 2 | 2 | 5 | 10 | 14 | 16 | 16 | 19 |
| $v_5 = 7, b_5 = 15$ | 0 | 2 | 2 | 5 | 10 | 14 | 16 | 16 | 19 |

Para que el algoritmo indique qué objetos hay que seleccionar para obtener el beneficio máximo, podemos llamar a una función que tenga como datos de entrada la tabla construida y los volúmenes de los objetos, y nos de como salida un vector *objetos* de ceros y unos, en el que un uno significa que hay que incluir el objeto para obtener el beneficio máximo. Esta función, empezando por la ultima casilla de la tabla, va comprobando a cada paso si el valor de la casilla coincide con el de la casilla de la fila superior, lo que indica que no se ha incluido el objeto *i*. Si no coinciden se sabe que el objeto se ha incluido y se pasa a comprobar la casilla correspondiente a la reducción de volumen que ha supuesto incluir el objeto.

```

fun ObjetosMochila(vol: vector, M:Tabla, n:entero, V:entero, objetos: Vector)
  var
    i,W: entero
  fvar
    W ← V
  para i ← n hasta 1 incremento -1 hacer
    si M[i,W] = M[i-1,W] entonces
      objetos[i] ← 0
    sino
      objetos[i] ← 1
      W ← W - vol[i]
    fsi
  fpara
ffun

```

5.6 Multiplicación asociativa de matrices

Este problema consiste en calcular la matriz producto *M* de *N* matrices. El producto de una matriz *A* de dimensiones $m \times n$ por una matriz *B* de dimensiones $n \times p$ es una matriz

C de dimensión $m \times p$ cuyas componentes son:

$$c_{ij} = \sum_{k=1}^n a_{ik} b_{kj}, 1 \leq i \leq m, 1 \leq j \leq p$$

Podemos observar que se requieren $m \times n \times p$ productos escalares para realizar la multiplicación de dos matrices.

El producto de matrices no es conmutativo, por lo que cuando queremos calcular el producto de una secuencia de matrices no podemos alterar su orden. Pero sí es asociativo, cumpliendo $(AB)C = A(BC)$. Esta propiedad nos permite agrupar una secuencia de matrices para multiplicar de distintas formas, y en función de la forma en que agrupemos las matrices el número de multiplicaciones puede variar mucho. Por ejemplo, si consideramos que tenemos las siguientes matrices:

| Matriz | Dimensiones |
|--------|---------------|
| A | 60×2 |
| B | 2×30 |
| C | 30×5 |
| D | 5×20 |

El número de multiplicaciones escalares necesarias para calcular el producto ABCD en función de la forma de agrupar los productos de matrices es el que muestra la siguiente tabla:

| Producto | Coste | Cálculo |
|------------|-------|--|
| $((AB)C)D$ | 18600 | $60 \times 2 \times 30 + 60 \times 30 \times 5 + 60 \times 5 \times 20$ |
| $A((BC)D)$ | 2900 | $2 \times 30 \times 5 + 2 \times 5 \times 20 + 60 \times 2 \times 20$ |
| $(AB)(CD)$ | 42600 | $60 \times 2 \times 30 + 30 \times 5 \times 20 + 60 \times 30 \times 20$ |
| $A(B(CD))$ | 6600 | $30 \times 5 \times 20 + 2 \times 30 \times 20 + 2 \times 20 \times 60$ |
| $(A(BC))D$ | 6900 | $2 \times 30 \times 5 + 60 \times 2 \times 5 + 60 \times 5 \times 20$ |

Podemos ver que las diferencias en el número de multiplicaciones escalares, y por tanto en el coste, son muy grandes dependiendo de la forma de agrupar los productos de matrices. El objetivo de este problema es buscar un algoritmo que halle el número mínimo de multiplicaciones escalares con el que se puede realizar el producto de una secuencia de matrices.

Veamos como plantear las ecuaciones de recurrencia para resolver el problema. Suponemos que tenemos una secuencia de matrices M_1, M_2, \dots, M_n para multiplicar. Las dimensiones de cada matriz M_i de la secuencia son $d_{i-1} \times d_i$ (la primera dimensión de la matriz M_i debe coincidir con la segunda de la matriz M_{i-1} para que sea posible realizar el producto). Sea $E(i, j)$ el número mínimo de multiplicaciones escalares que se necesitan

para realizar el producto de las matrices M_i, M_{i+1}, \dots, M_j , $1 \leq i \leq j \leq n$. Una posible asociación para realizar este producto la podemos representar por

$$(M_i M_{i+1} \cdots M_k) (M_{k+1} \cdots M_j)$$

El número de productos escalares que requiere realizar este producto es el número de productos para multiplicar las matrices del primer paréntesis, $E(i, k)$, más el número necesario para las matrices del segundo paréntesis, $E(k+1, j)$, más los productos necesarios para multiplicar las matrices resultantes de estos dos productos. Como la matriz resultante de $(M_i M_{i+1} \cdots M_k)$ tiene dimensiones (d_{i-1}, d_k) y la matriz resultante de $(M_{k+1} \cdots M_j)$ tiene dimensiones (d_k, d_j) , el número de productos escalares para estas dos matrices es $d_{i-1} d_k d_j$. Debemos plantear el problema de forma que se elija k para minimizar el número total de productos escalares. Llegamos al caso base cuando $i = j$, es decir, tenemos una única matriz y por tanto el número de productos es 0.

A partir de este razonamiento llegamos a la ecuación de recurrencia para el problema:

$$E(i, j) = \begin{cases} \min_{i \leq k < j} \{E(i, k) + E(k+1, j) + d_{i-1} d_k d_j\} & \text{si } i < j \\ 0 & \text{si } i = j \end{cases}$$

En este problema se cumple el principio de optimalidad ya que si $E(i, j)$ es mínimo, también han de serlo $E(i, k)$ y $E(k+1, j)$.

Para resolver esta ecuación siguiendo el esquema de programación dinámica construimos una tabla *mult* que almacene los resultados parciales de $E(i, j)$. Como ocurría en el problema del viaje por el río (Sección 5.4) sólo necesitamos la parte superior por encima de la diagonal principal. Entonces, hacemos como en aquel problema, rellenando la tabla por diagonales. Las diagonales se numeran desde $d = 1$, que está justo encima de la diagonal principal, hasta $d = n - 1$. La fila de los elementos de cada diagonal varía entre $i = 1$ y $i = n - d$, y la columna se calcula como $j = i + d$.

Podemos ahora escribir un algoritmo que rellena esta tabla aplicando la ecuación anterior:

```

tipo Tabla = matriz[1..N,1..N] de entero
tipo Vector = matriz[0..N] de entero
fun MultMatrices(d: Vector, N: entero, mult: Tabla)
    var
        i,diag: entero
    fvar
    para i  $\leftarrow$  1 hasta N hacer
        mult[i,i]  $\leftarrow$  0
    fpara
    para diag  $\leftarrow$  1 hasta N-1 hacer
        para i  $\leftarrow$  1 hasta N-diag hacer

```

```
mult[i,i+diag] ← MinMultiple(mult,d,i,i+diag)
```

fpara

fpara

ffun

El algoritmo recibe como parámetros de entrada el número de matrices para multiplicar, N , y un vector d que contiene las dimensiones de las matrices, es decir la dimensión de la matriz i es $d[i - 1] \times d[i]$. El parámetro de salida es la tabla $mult$ con el número mínimo de multiplicaciones para cada caso. La solución al problema es el valor de $mult[1,n]$. La función $MinMultiple$, que calcula el mínimo entre múltiples valores, es la siguiente:

```
fun MinMultiple(mult: Tabla, d: Vector, i: entero, j: entero): entero
    var
        k, minimo: entero
    fvar
        minimo ← ∞
    para k ← i hasta j-1 hacer
        minimo ← min(minimo, mult[i,k] + mult[k+1,j] + d[i-1]*d[k]*d[j])
    fpara
    dev minimo
ffun
```

Para calcular el coste temporal observamos que en el algoritmo hay dos bucles anidados, cada uno de ellos de orden $O(N)$, dentro de los cuales se llama a la función $MinMultiple$, que también es de orden $O(N)$. Por tanto la complejidad temporal está en $O(N^3)$. Por otra parte, el coste espacial está en $O(N^2)$ ya que construimos una tabla $N \times N$.

Como en otros problemas de este capítulo, podemos considerar una versión alternativa en la que no sólo buscamos el número mínimo de multiplicaciones, sino también el parentizado que nos permite obtenerlas. Para conseguir esto basta añadir una nueva tabla pos que almacena la posición en la que se coloca el paréntesis para conseguir el producto óptimo. Al final del algoritmo la función *EscribeParentizado* escribe la asociación de productos de matrices que lleva a minimizar los productos escalares.

tipo Tabla = matriz[1..N,1..N] de entero

tipo Vector = matriz[0..N] de entero

fun MultMatrices2(d: Vector, N: entero, mult: Tabla, pos: Tabla)

var

 i,diag: entero

fvar

para i ← 1 hasta N **hacer**

 mult[i,i] ← 0

fpara

```

para diag  $\leftarrow 1$  hasta N-1 hacer
  para i  $\leftarrow 1$  hasta N-diag hacer
    MinMultiple2(mult,d,i,i+diag, minimo, posicion)
    mult[i,i+diag]  $\leftarrow$  minimo
    pos[i,i+diag]  $\leftarrow$  posicion
  fpara
fpara
  EscribeParentizado(pos, 1, N)
ffun

```

La función *MinMultiple* también se modifica para registrar la posición del paréntesis:

```

fun MinMultiple2(mult: Tabla, d: Vector, i: entero, j: entero, minimo: entero,
                   pos: entero)
  var
    k, tmp: entero
  fvar
    minimo  $\leftarrow \infty$ 
    pos  $\leftarrow i$ 
  para k  $\leftarrow i$  hasta j-1 hacer
    tmp  $\leftarrow$  mult[i,k] + mult[k+1,j] + d[i-1]*d[k]*d[j]
    si tmp < minimo entonces
      minimo  $\leftarrow$  tmp
      pos  $\leftarrow$  k
    fsi
  fpara
ffun

```

Ahora podemos escribir una función que va colocando los paréntesis en los lugares indicados en *pos*:

```
fun EscribeParentizado(pos: Tabla, i: entero, j: entero)
```

```

  var
    k: entero
  fvar
    k  $\leftarrow$  pos[i,j]
    Imprimir "("
    EscribeParentizado(pos,i,k)
    EscribeParentizado(pos,k+1,j)
    Imprimir ")"
ffun

```

```

  Si i=j entonces
    Imprimir "M", imprimir i
  Si no
    K<--- pos[i,j]
    Imprimir "("
    EscribeParentizado(pos,i,k)
    EscribeParentizado(pos,k+1,j)
    Imprimir ")"
  Fsi

```

Este algoritmo *EscribeParentizado* tiene un coste lineal con el número de matrices *N*.

5.7 Camino de coste mínimo entre nodos de un grafo dirigido

Consideramos un grafo dirigido G compuesto por N nodos $\{1, 2, \dots, N\}$ y un conjunto de aristas A que tienen asociada una longitud no negativa. El objetivo de este problema es calcular la longitud del camino más corto entre cada par de nodos del grafo. Anteriormente, en el capítulo dedicado a los algoritmos voraces hemos buscado el camino más corto entre un nodo y los restantes nodos. Como vimos, el algoritmo de Dijkstra resolvía este problema con una complejidad $O(N^2)$, por lo que aplicado a la búsqueda del camino más corto entre cada par de nodos del grafo su complejidad es de orden $O(N^3)$. Ahora nos planteamos un algoritmo alternativo basado en programación dinámica para resolver el problema. Como veremos, la complejidad del algoritmo resultante, conocido como *algoritmo de Floyd*[Flo62], es también del orden $O(N^3)$.

En este problema se cumple el principio de optimalidad, ya que si el camino mínimo entre un nodo i y un nodo j incluye a un nodo k entonces los caminos entre i y k y entre k y j deben ser mínimos.

Veamos como podemos plantear la ecuación de recurrencia. Sea $M(i, j, k)$ el coste mínimo de ir del nodo i al nodo j pudiendo utilizar como nodos intermedios los que están entre 1 y k . El camino más corto entre los vértices i y j cuando consideramos un vértice intermedio k puede corresponder a una de las dos siguientes posibilidades:

- Puede pasar por k , lo que nos lleva a buscar los caminos óptimos entre i y k , y entre k y j . Aceptamos que ninguno de los caminos intermedios pasará por k , ya que en este caso habría un ciclo que se podría eliminar.
- Puede no pasar por k , y entonces hay que considerar el mejor camino de i a j utilizando sólo como posibles vértices intermedios los que están entre 1 y $k - 1$.

El caso base se alcanza con $k = 0$ y corresponde a ir de i a j sin pasar por nodos intermedios. Si $i = j$ este coste es 0, y si no coinciden es la longitud de la arista entre i y j .

A partir de estas ideas planteamos la ecuación de recurrencia de la siguiente forma:

$$M(i, j, k) = \begin{cases} 0 & \text{si } k = 0 \text{ y } i = j \\ A[i, j] & \text{si } k = 0 \text{ y } i \neq j \\ \min(M(i, j, k - 1), \\ M(i, k, k - 1) + M(k, j, k - 1)) & \text{si } k > 0 \end{cases}$$

El coste mínimo entre dos vértices del grafo i y j corresponde a buscar el camino mínimo considerando como intermedio a cualquiera de los otros nodos $M(i, j, n)$.

Veamos ahora las estructuras que necesitamos para almacenar los resultados intermedios. En primer lugar podemos observar que para calcular los datos cuando se incluye

el nodo k , es decir, la tabla correspondiente a $M(i, j, k)$, sólo se necesitan los datos correspondientes a haber incluido el nodo $k - 1$. Esto nos permite usar una única tabla $N \times N$, cuyos datos se van actualizando a medida que se incluyen nuevos nodos. Si se requirieran los datos intermedios necesitaríamos una tabla para cada k , lo que nos llevaría a una complejidad espacial de $O(N^3)$. Concretamente, para calcular $M(i, j, k)$ necesitamos $M(i, j, k - 1)$, $M(i, k, k - 1)$ y $M(k, j, k - 1)$. La tabla se inicializa con los datos correspondientes al caso base $k = 0$, que se corresponde con las longitudes de las aristas de grafo, es decir, su matriz de adyacencia.

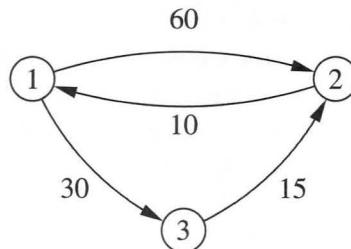
El algoritmo con el que se construye esta tabla para resolver el problema, como ya se ha indicado se conoce como *algoritmo de Floyd*, y tiene la siguiente forma:

```

tipo Tabla = matriz[1..N,1..N] de entero
fun Floyd(A: Tabla, N: entero, M: Tabla)
    var
        i, j, k: entero
    fvar
    para i  $\leftarrow$  0 hasta N hacer
        para j  $\leftarrow$  0 hasta N hacer
            M[i,j]  $\leftarrow$  A[i,j]
        fpara
    fpara
    para k  $\leftarrow$  1 hasta N hacer
        para i  $\leftarrow$  1 hasta N hacer
            para j  $\leftarrow$  1 hasta N hacer
                M[i,j]  $\leftarrow$  min(M[i,j], M[i,k] + M[k,j])
            fpara
        fpara
    fpara
ffun
```

El coste temporal de este algoritmo, con los tres bucles anidados, está en $O(N^3)$, mientras el coste espacial está en $O(N^2)$. Aunque la complejidad de los algoritmos de Dijkstra y Floyd es la misma, las operaciones del algoritmo de Floyd son más simples, lo que nos indica que probablemente le correspondan constantes ocultas más pequeñas, y por tanto en la práctica sea más rápido el algoritmo de Floyd. Sin embargo, para grafos poco densos, el algoritmo de Dijkstra puede optimizarse utilizando listas de distancias a nodos adyacentes para representar el grafo, lo que aumenta su velocidad de cómputo. Por lo tanto, Floyd es preferible para grafos densos, y Dijkstra para grafos dispersos.

Veamos como funciona el algoritmo en un pequeño ejemplo de 3 nodos. Consideraremos el siguiente grafo:



Entonces la tabla inicial que construye el algoritmo para $k = 0$ es la siguiente, que se corresponde con la matriz de adyacencia del grafo:

| | 1 | 2 | 3 |
|---|----------|----|----------|
| 1 | 0 | 60 | 30 |
| 2 | 10 | 0 | ∞ |
| 3 | ∞ | 15 | 0 |

Incluimos ahora el nodo 1 y construimos la tabla para $k = 1$:

| | 1 | 2 | 3 |
|---|----------|----|----|
| 1 | 0 | 60 | 30 |
| 2 | 10 | 0 | 40 |
| 3 | ∞ | 15 | 0 |

Vemos que ahora ya se ha encontrado un camino entre los nodos 2 y 3 que pasa por el nodo 1 y tiene una longitud de 40. Incluimos el siguiente nodo en el cálculo de los caminos mínimos y construimos la tabla para $k = 2$:

| | 1 | 2 | 3 |
|---|----|----|----|
| 1 | 0 | 60 | 30 |
| 2 | 10 | 0 | 40 |
| 3 | 25 | 15 | 0 |

Tenemos ahora un camino del nodo 3 al 1 que pasa por el nodo 2 y tiene coste 25. Por último construimos la tabla para $k = 3$:

| | 1 | 2 | 3 |
|---|----|----|----|
| 1 | 0 | 45 | 30 |
| 2 | 10 | 0 | 40 |
| 3 | 25 | 15 | 0 |

Vemos que al considerar los caminos que pasan por el nodo 3 hemos encontrado un camino más corto del nodo 1 al 2 con longitud 45.

Para resolver el problema de cuál es el camino o la secuencia de nodos que constituye el camino más corto, y no sólo su longitud, añadimos una tabla adicional *ruta*, que registra los nodos con los que se construye el camino mínimo. El algoritmo en este caso toma la siguiente forma:

```

tipo Tabla = matriz[1..N,1..N] de entero
fun Floyd(A: Tabla, N: entero, M: Tabla, ruta: Tabla)
    var
        i,j,k,tmp: entero
    fvar
    para i ← 0 hasta N hacer
        para j ← 0 hasta N hacer
            M[i,j] ← A[i,j]
            ruta[i,j] ← 0
    fpara
    fpara
    para k ← 1 hasta N hacer
        para i ← 1 hasta N hacer
            para j ← 1 hasta N hacer
                tmp ← M[i,k] + M[k,j]
                si tmp < M[i,j] entonces
                    M[i,j] ← tmp
                    ruta[i,j] ← k
                fsi
    fpara
    fpara
    fpara
ffun
```

Inicializamos la tabla *ruta* a 0, un nodo que no existe, indicando que por defecto se consideran caminos directos.

Ahora podemos imprimir el camino mínimo a seguir entre dos nodos dados de la siguiente forma:

```

fun VerRutas(N: entero, M: Tabla, ruta: Tabla)
    var
        i,j: entero
    fvar
    para i ← 0 hasta N hacer
        para j ← 0 hasta N hacer
```

```

si M[i,j] ≠ ∞ entonces
    Imprimir('ruta de ',i,' a ',j)
    Imprimir(i)
    ImprimeRutaRec(ruta,i,j)
    Imprimir(j)
fsi
fpara
fpara
ffun

```

La función *ImprimeRuta* recorre la tabla *ruta* comprobando si el camino entre *i* y *j* es directo (hay un 0 en la tabla) o pasa por algún nodo intermedio *k*. En el último caso imprime recursivamente el subcamino anterior y posterior a *k*:

```

fun ImprimeRutaRec(ruta: Tabla, i: entero, j: entero)
    var
        k: entero
    fvar
        k ← ruta[i,j]
    si k ≠ 0 entonces
        ImprimeRutaRec(ruta,i,k)
        Imprimir(k)
        ImprimeRutaRec(ruta,k,j)
    fsi
ffun

```

5.8 Distancia de edición

Se consideran dos cadenas de caracteres, *X* e *Y*, de un alfabeto finito. La cadena *X* = x_1, x_2, \dots, x_n tiene longitud *n*, y la cadena *Y* = y_1, y_2, \dots, y_m tiene longitud *m*. La cadena *X* se puede transformar en la cadena *Y* realizando los siguientes tipos de cambios:

- Borrar un carácter de *X*.
- Insertar uno de los caracteres de *Y* en *X*.
- Sustituir un carácter de *X* por uno de los de *Y*.

Por ejemplo, consideremos las dos siguientes cadenas:

$$\begin{aligned} X &= aabbcb \\ Y &= bbbcba \end{aligned}$$

Podemos pasar de X a Y haciendo los siguientes cambios:

| | |
|-----------------|---|
| aabbcb → babbcb | cambiando la ‘a’ de la posición 1 por ‘b’ |
| babbcb → bbbbcb | cambiando la ‘a’ de la posición 2 por ‘b’ |
| bbbbcb → bbbccb | cambiando la ‘b’ de la posición 4 por ‘c’ |
| bbbccb → bbbccb | cambiando la ‘c’ de la posición 5 por ‘b’ |
| bbbcbb → bbbcba | cambiando la ‘b’ de la posición 6 por ‘a’ |

Hemos necesitado 5 cambios para transformar la cadena X en la cadena Y . Sin embargo, esta transformación puede hacerse en menos pasos:

| | |
|-----------------|---|
| aabbcb → abbcbb | borrando la ‘a’ de la posición 1 |
| abbcbb → bbbccb | cambiando la ‘a’ de la posición 1 por ‘b’ |
| bbbccb → bbbcba | insertando el carácter ‘a’ en la posición 6 |

El objetivo de este problema es encontrar el número mínimo de cambios para transformar la cadena X en la cadena Y .

Veamos como podemos formular la ecuación de recurrencia del problema. Sea $C(i, j)$ el número mínimo de cambios necesarios para transformar la cadena x_1, x_2, \dots, x_i en la cadena y_1, y_2, \dots, y_j .

Consideramos la posición i de la cadena X y la posición j de la cadena Y . Si elegimos realizar una operación de borrado en la posición i de X entonces el número de transformaciones sería $1 + C(i - 1, j)$. Si elegimos insertar y_j en la posición j entonces el número de cambios sería $1 + C(i, j - 1)$. Por último, si elegimos sustituir x_i por y_j en la posición j , el número de transformaciones será $1 + C(i - 1, j - 1)$ si x_i y y_j son distintos y $C(i - 1, j - 1)$ si son iguales.

Los casos base se corresponden a estar en la posición cero de una de las cadenas. Es decir, cuando se tiene la cadena objetivo y sobran caracteres de la de origen que hay que borrar ($C(i, 0) = i$), en cuyo caso hay que realizar operaciones de borrado, o cuando no quedan caracteres en la cadena de origen pero faltan en la cadena objetivo ($C(0, j) = j$), en cuyo caso hay que realizar operaciones de inserción.

Tenemos por tanto, la siguiente ecuación de recurrencia:

$$C(i, j) = \begin{cases} i & \text{si } j = 0 \\ j & \text{si } i = 0 \\ 1 + \min\{C(i - 1, j), C(i, j - 1), C(i - 1, j - 1)\} & \text{si } i \neq 0, j \neq 0, x_i \neq y_j \\ \min\{C(i - 1, j) + 1, C(i, j - 1) + 1, C(i - 1, j - 1)\} & \text{si } i \neq 0, j \neq 0, x_i = y_j \end{cases}$$

Siguiendo el esquema de la programación dinámica construimos una tabla $C[n, m]$. Después de la inicialización de los casos base, cada casilla $C[i, j]$ de esta tabla se construye a partir de los resultados de las casillas $C[i - 1, j - 1]$ y $C[i - 1, j]$ de la fila anterior, y de la casilla $C[i, j - 1]$ de la columna anterior y la misma fila. Por tanto, rellenamos la tabla por filas de arriba abajo y de izquierda a derecha. La solución al problema es el valor de la casilla $C[n, m]$.

El algoritmo puede escribirse de la siguiente forma:

```

tipo Tabla = matriz[0..n,0..m] de entero
fun DistanciaEdicion(X: Vector[1..n] de caracter, Y: Vector[1..m] de caracter,
                      n,m: entero, C: Tabla)
var
    i,j,tmp: entero
fvar
para i ← 0 hasta n hacer
    C[i,0] ← i
fpara
para j ← 0 hasta m hacer
    C[0,j] ← j
fpara
para i ← 1 hasta n hacer
    para j ← 1 hasta m hacer
        tmp ← min(1 + C[i-1,j], 1 + C[i,j-1])
        si X[i] = Y[j] entonces
            C[i,j] ← min(tmp, C[i-1,j-1])
        sino
            C[i,j] ← min(tmp, C[i-1,j-1]+1)
        fsi
    fpara
fpara
ffun

```

El coste temporal del algoritmo viene dado por los dos bucles anidados que se utilizan para construir la tabla, y es por tanto $O(nm)$. El coste temporal, dado por el tamaño de la tabla, es el mismo.

Para poder responder no sólo cuál es el número mínimo de cambios, sino también qué cambios son los que hay que realizar, necesitamos especificar en cada transformación la operación de que se trata (borrado, inserción o sustitución), la posición en la que se realiza y el carácter afectado. Estas estructuras las podemos almacenar en un vector *transforma* de longitud $n \times m$, una cota superior al número de transformaciones a realizar.

```

tipo Trans =
registro
    ope: Vector[1..30] de caracter
    pos: entero
    car: caracter
fregistro

```

```

fun IdentificarTrans(X: Vector[1..n] de caracter, Y: Vector[1..m] de caracter,
    n,m: entero, C: Tabla, transforma: Vector[1..nm] de Trans)
var
    i,j,k: entero
fvar
    k ← C[n,m]
mientras k > 0 hacer
    caso de
        C[i,j] = C[i-1,j]+1 hacer
            transforma[k].ope ← “borrado”
            transforma[k].pos ← j+1
            k ← k - 1
            i ← i - 1
        C[i,j] = C[i,j-1]+1 hacer
            transforma[k].ope ← “insercion”
            transforma[k].pos ← j
            transforma[k].car ← Y[j]
            k ← k - 1
            j ← j - 1
        C[i,j] = C[i-1,j-1]+1 hacer
            transforma[k].ope ← “sustitucion”
            transforma[k].pos ← j
            transforma[k].car ← Y[j]
            k ← k - 1
            j ← j - 1
            i ← i - 1
        C[i,j] = C[i-1,j-1] hacer
            j ← j - 1
            i ← i - 1
    fcaso
fmientras
ffun

```

5.9 Ejercicios propuestos

1. Se pide un algoritmo que resuelva el problema del viajante de comercio con programación dinámica. En este problema un viajante tiene que recorrer todo un conjunto de ciudades visitando cada una de ellas una sola vez y regresando al punto de partida. Se conoce la distancia entre cada par de ciudades y el objetivo

es hallar la longitud del recorrido más corto que puede hacer el viajante. ¿Cuál es el coste temporal y espacial del algoritmo?

2. Si en el ejercicio anterior se pide que el algoritmo indique también el recorrido a realizar y no sólo su longitud, ¿cómo se ve afectada la complejidad espacial del algoritmo?
3. Se pide resolver el problema de la mochila, pero considerando que en lugar de disponer de n objetos distintos, se dispone de n tipos de objetos, de manera que podemos seleccionar tantos objetos como se desee de un tipo determinado. El problema se puede formular sustituyendo la representación 0 ó 1 que en la formulación original utilizábamos para indicar si un objeto se incluía en la mochila, por valores enteros que indican el número de unidades de cada tipo que se incluyen. El objetivo vuelve a ser maximizar la suma de los beneficios de los objetos introducidos en la mochila, de manera que no se supere la capacidad de la mochila.
4. Se considera un conjunto de estaciones de trenes para las que se conoce el tiempo del recorrido del tren que conecta algunos de los pares de estaciones. Se pide un algoritmo que, dadas una estación origen y una destino, busque el trayecto que puede pasar por estaciones intermedias minimizando el tiempo total. No todas las estaciones tienen porque estar conectadas directamente por un tren. Se supone que el tiempo de cambio de tren es despreciable.
5. Se consideran dos equipos de fútbol X e Y que compiten por ser el primero en alcanzar n victorias. Se supone que los resultados de los partidos son independientes, que no hay empates, y que para cualquier partido la probabilidad de que lo gane el equipo X es p , y de que lo gane el equipo Y es $q = 1 - p$. Se pide un algoritmo que calcule la probabilidad de que el equipo X gane la competición.
6. Se tiene un conjunto de n tareas que deben ser procesadas en un sistema compuesto por dos procesadores X e Y . La ejecución de cada tarea i requiere un tiempo x_i en el procesador X e y_i en el procesador Y . Debido a los distintos tipos de operaciones que requieren las tareas, puede ocurrir que para una tarea i se cumpla $x_i > y_i$, mientras que para otra tarea j , $x_j < y_j$. Se pide un algoritmo que asigne las tareas a los procesadores de forma que se minimice el tiempo requerido para realizarlas todas.
7. Se pide resolver el problema de la devolución de cambio pero suponiendo que de cada tipo de moneda x_i sólo se dispone de una cantidad limitada de monedas m_i .
8. Se considera un grafo dirigido a cuyas aristas se han asignado valores positivos. Se pide calcular el número de caminos de coste mínimo para cada par de vértices del grafo.

9. Se tiene un conjunto de n tipos de sellos diferentes, de los que se dispone de una cantidad ilimitada de cualquier tipo. Se pide un algoritmo que calcule el número de formas diferentes de franquear una carta cuya tarifa de envío es C , si se quiere utilizar el franqueo exacto (el valor de los sellos debe ser exactamente C).
10. Se pide un algoritmo de programación dinámica para calcular la función de *Ackermann* definida de la siguiente forma:

$$Ackermann(m, n) = \begin{cases} n + 1 & \text{si } m = 0 \\ Ackermann(m - 1, 1) & \text{si } m > 0, n = 0 \\ Ackermann(m - 1, Ackermann(m, n - 1)) & \text{si } m > 0, n > 0 \end{cases}$$

5.10 Notas bibliográficas

Este tema se describe detalladamente en diversos textos clásicos de algorítmica, como el de Brassard y Bratley [BB06]. También se trata en diversos libros de carácter más práctico [Gue00, MOOMV03, GMCLMPGC03] en los que se pueden encontrar diversos problemas resueltos utilizando el esquema de la programación dinámica. Así en [Gue00] se pueden encontrar entre otros, distintas versiones de los problemas del viajante de comercio y de la mochila, así como el del trayecto óptimo en tren y la función de Ackermann. En [MOOMV03] se resuelven entre otros los problemas de la competición de equipos, el procesamiento de tareas y el franqueo de una carta. El estudio sistemático de la programación dinámica se inició en 1955 por R. Bellman [Bel57], que proporcionó una sólida base matemática al tema.

Capítulo 6

Vuelta atrás

En este capítulo se presenta el esquema de Vuelta Atrás o retroceso. Este esquema se aplica a problemas en los que sólo podemos recurrir a una búsqueda exhaustiva, recorriendo el espacio de todas las posibles soluciones hasta que encontremos una de ellas o hasta que hayamos explorado todas las opciones, concluyendo así que no existe la solución buscada. Puesto que esta búsqueda exhaustiva es muy costosa, es importante aplicar el conocimiento disponible sobre el problema para terminar la exploración de un camino tan pronto como se sepa que dicho camino no puede alcanzar una solución. El esquema que se presenta aquí refleja estas ideas. La descripción del esquema general se exemplifica detalladamente sobre el problema de colocar N reinas en un tablero de ajedrez de forma que no se ataquen. Después se presenta la resolución de otros problemas aplicando el esquema de vuelta atrás: el problema del coloreado de grafos, la búsqueda de ciclos hamiltonianos, de subconjuntos de suma dada, el reparto equitativo de activos, el robot en busca de un tornillo, y la asignación de cursos en una escuela. Se describen los elementos del esquema y una cota superior al coste que pueden tener los algoritmos resultantes.

Este capítulo debe estudiarse después del capítulo dedicado a las estructuras de datos avanzadas en el que se estudian los grafos y los algoritmos para recorrerlos, ya que se utilizan en la implementación del esquema. También es conveniente estudiar previamente el capítulo dedicado a los algoritmos voraces para poder decidir al considerar un nuevo problema si es posible aplicar el esquema voraz, siempre más eficiente, o si hay que recurrir a otro esquema.

6.1 Planteamiento general

En muchos casos no existe un algoritmo específico que resuelva un problema de forma sistemática. Entonces tenemos que recurrir a una búsqueda exhaustiva de las posibles soluciones. Muchos problemas se pueden formular como la búsqueda de un determinado nodo, un camino o algún tipo de patrón en el grafo que representa el espacio de

soluciones del problema. Si el grafo es infinito o incluso si es muy grande, no tiene sentido que el programa lo construya de forma explícita, para luego aplicar las técnicas de búsqueda que se han visto. En estos casos se trabaja con un grafo implícito. Se trata de un grafo para el que disponemos de una descripción de sus nodos y aristas, de forma que se van construyendo sólo aquellas partes del grafo a las que llega el recorrido. En éste y en el siguiente capítulo veremos dos de las principales formas de realizar recorridos en un grafo implícito para resolver problemas.

El esquema de vuelta atrás realiza un recorrido en profundidad del grafo implícito de un problema, podando aquellas ramas para las que el algoritmo puede comprobar que no pueden alcanzar una solución al problema. Las soluciones se construyen de forma incremental, de forma que a un nodo del nivel k del árbol le corresponderá una parte de la solución construida en los k pasos dados en el grafo para llegar a dicho nodo, y que llamamos solución o secuencia k -prometedora. Si la solución parcial construida en un nodo no se puede extender o completar, decimos que se trata de un *nodo de fallo*. Cuando se da esta situación, el algoritmo *retrocede* hasta un nodo del que quedan ramas pendientes de ser exploradas. En su forma más básica, la vuelta atrás es similar a un recorrido en profundidad dentro de un grafo dirigido. El grafo suele ser un árbol, o en todo caso un grafo sin ciclos. A medida que progresa el recorrido por el grafo implícito se van construyendo soluciones parciales. Un recorrido tiene éxito si se llega a completar una solución. Por el contrario, no tiene éxito si en alguna etapa la solución parcial construida no se puede seguir completando. En este caso el recorrido vuelve atrás como en un recorrido en profundidad, eliminando los elementos que se hayan añadido en cada fase. Cuando se llega a algún nodo que tiene algún vecino sin explorar prosigue el recorrido de búsqueda de una solución a partir de él. El objetivo del algoritmo puede ser encontrar una solución o encontrar todas las posibles soluciones al problema. En el primer caso el algoritmo se detiene al encontrar la primera solución, evitando así la construcción del resto del grafo del problema, lo que puede suponer una gran ganancia en eficiencia.

Los elementos principales del esquema de vuelta atrás son:

- *IniciarExploraciónNivel()*: recoge todas las opciones posibles en que se puede extender la solución k -prometedora.
- *OpcionesPendientes()*: comprueba que quedan opciones por explorar en el nivel.
- *SoluciónCompleta()*: comprueba que se haya completado una solución al problema.
- *ProcesarSolución()*: representa las operaciones que se quieran realizar con la solución, como imprimirla o devolverla al punto de llamada.
- *Completable()*: comprueba que la solución k -prometedora se puede extender con la opción elegida cumpliendo las restricciones del problema hasta llegar a completar una solución.

Con estos elementos, el esquema general de vuelta atrás que busca todas las soluciones puede formularse de la siguiente forma:

```
fun VueltaAtras (v: Secuencia, k: entero)
  {v es una secuencia k-prometedora}
  IniciarExploraciónNivel(k)
  mientras OpcionesPendientes(k) hacer
    extender v con siguiente opción
    si SoluciónCompleta(v) entonces
      ProcesarSolución(v)
    sino
      si Completable (v) entonces
        VueltaAtras(v, k+1)
      fsi
    fmientras
  fsi
ffun
```

Vemos que el esquema refleja la construcción incremental de las soluciones. Su aplicación requiere que la solución al problema pueda expresarse como una secuencia $[x_1, x_2, \dots, x_n]$ en la que cada componente x_i se elige en una de las etapas del algoritmo. Cada etapa corresponde a un nivel del árbol de expansión. Cuando una de estas secuencias se completa, si sólo se buscaba una solución, se ha alcanzado una solución al problema y el algoritmo termina. Mientras la secuencia no está completa, el árbol se expande para todas las componentes válidas de la siguiente etapa: las $k + 1$ prometedoras. Por tanto, el primer paso para la aplicación de este esquema consiste en definir la estructura que va a representar las soluciones al problema, y el significado de sus componentes.

En el caso de que sólo se necesite una solución, se añade un parámetro booleano *encontrado*, cuyo valor inicial es falso, y que se hace cierto al encontrar la primera solución. El esquema puede adoptar entonces la siguiente forma:

```
fun VueltaAtras (v: Secuencia, k: entero, encontrado: booleano)
  {v es una secuencia k-prometedora}
  IniciarExploraciónNivel(k)
  mientras OpcionesPendientes(k)  $\wedge \neg$  encontrado hacer
    extender v con siguiente opción
    si SoluciónCompleta(v) entonces
      ProcesarSolución(v)
      encontrado  $\leftarrow$  cierto
    sino
```

si Completable (v) **entonces**

VueltaAtras(v, k+1, encontrado)

fi

fmiéntras

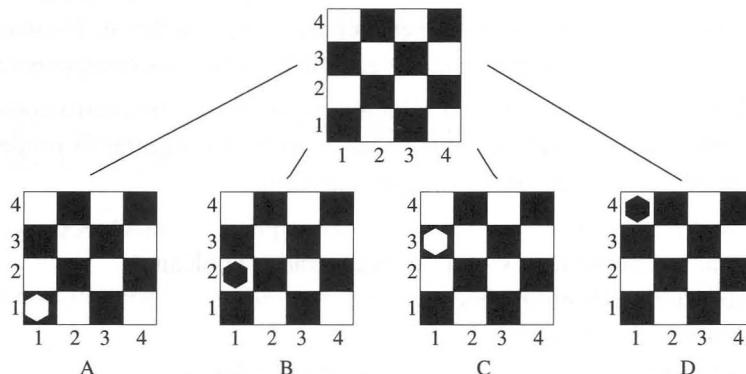
fi

ffun

El coste del caso peor de los algoritmos de búsqueda exhaustiva es del orden del tamaño del espacio de búsqueda. Las funciones de poda que utilicemos reducen el coste, aunque con frecuencia no es posible evaluar de qué forma, porque la poda depende de los datos concretos a los que se aplica el algoritmo. Por ello, en la mayoría de los casos daremos una cota superior al coste que calcularemos en función del tamaño del árbol de soluciones.

Un ejemplo típico de aplicación de este esquema es el problema de las N reinas de ajedrez. Este problema consiste en colocar N reinas en un tablero de ajedrez $N \times N$ de forma que ninguna reina ataque a otra. Esto significa que no puede haber dos reinas en una misma fila, ni en una misma columna o diagonal. En este caso podemos partir del hecho de que cada reina va a estar en una columna distinta, es decir la columna de la reina c_i va a ser i , y considerar que la solución es una asignación de la fila que corresponde a cada reina. Por tanto, las soluciones son un vector de enteros en el que cada componente f_i representa la fila asignada a la reina i .

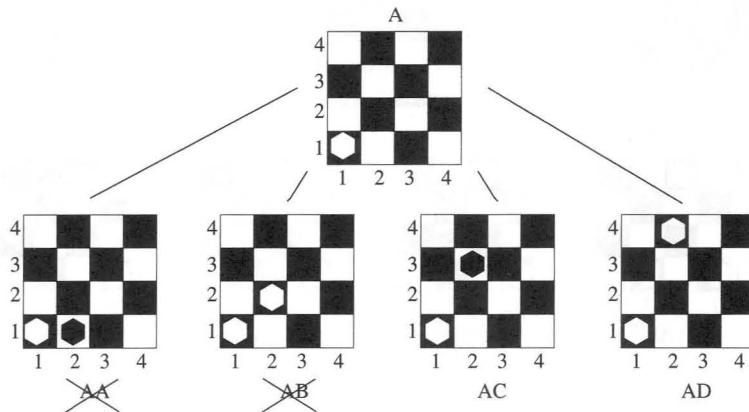
Para entender la aplicación del esquema de vuelta atrás a este problema vamos a considerar un caso muy sencillo con una tablero 4×4 en el que queremos colocar 4 reinas. La siguiente figura muestra la situación inicial:



Partimos del tablero vacío, y construimos el primer nivel de nodos del árbol. La solución parcial inicial es un vector vacío. Este vector se expande con un nuevo elemento para el que tenemos cuatro posibilidades: las cuatro filas posibles. Por tanto, el nodo inicial del árbol se expande en 4 nodos, correspondientes a las soluciones parciales $[1, \dots, \dots]$,

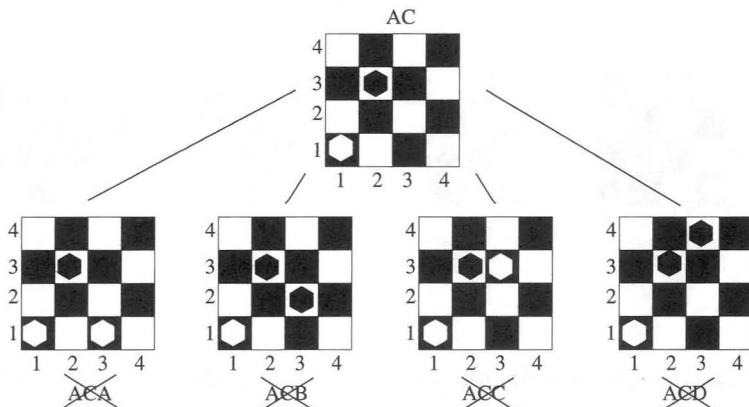
[2, __, __], [3, __, __], [4, __, __]. Como se trata de la primera reina que se coloca, no hay conflictos y todas las alternativas son completables.

El siguiente paso es expandir cada uno de los nodos *A*, *B*, *C* y *D* de la figura anterior colocando una nueva reina en la segunda columna. El algoritmo expande en primer lugar el nodo *A*, almacenando los restantes para el caso de que la opción elegida no permita completar una solución. La siguiente figura muestra las posibles expansiones del nodo *A*:

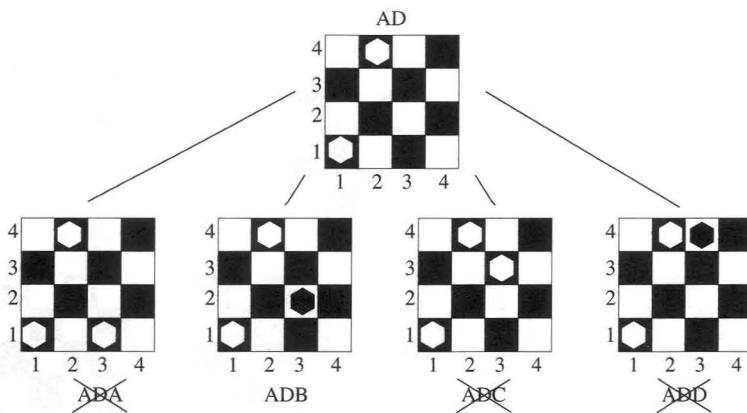


Podemos observar que el nodo *AA* corresponde a una solución parcial no completable (tiene dos reinas en la misma fila), por lo que se poda y ya no se expande, lo que supone un ahorro frente a un recorrido exhaustivo en profundidad. Lo mismo ocurre con el nodo *AB* que tiene dos reinas en la misma diagonal. Nos quedan los nodos *AC* y *AD*. Seguimos la exploración expandiendo el nodo *AC* y reservamos *AD* para un posible retroceso en la exploración.

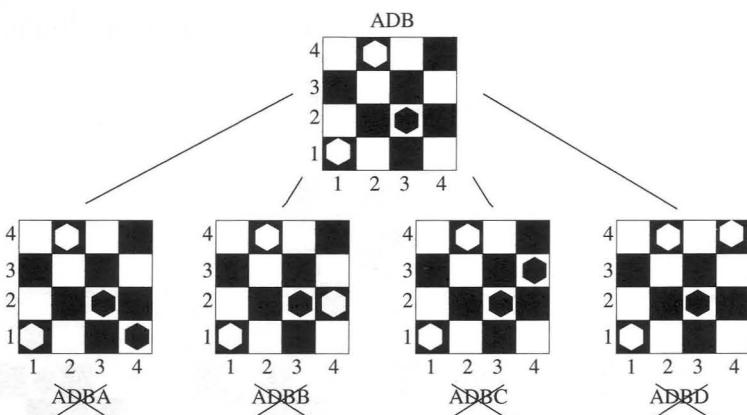
La siguiente figura muestra la expansión del nodo *AC*:



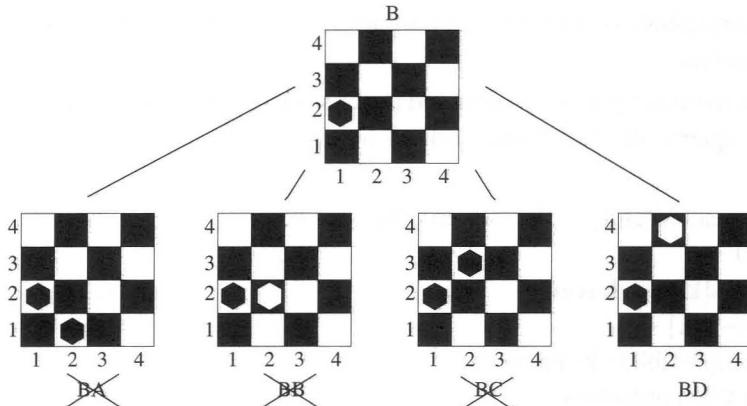
En este caso vemos que ninguna de las expansiones es completable, ya en todas ellas las reinas se atacan de una u otra forma. Por ello, **volvemos atrás** al nodo *AD* que estaba pendiente de explorar en el nivel dos. La siguiente figura muestra la expansión del nodo *AD*:



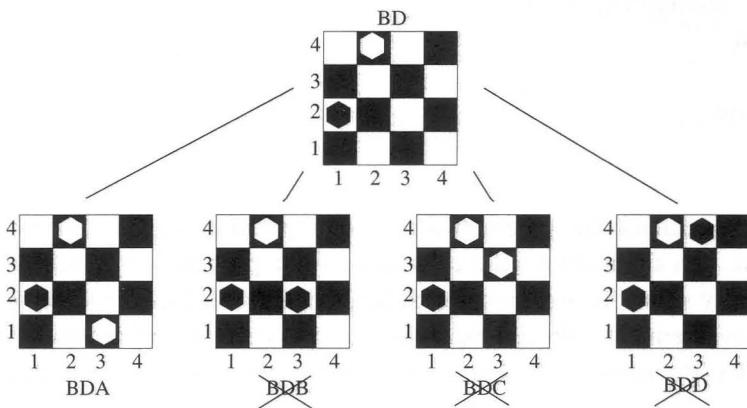
En este caso sólo uno de los nodos, *ADB*, representa una solución parcial completable. Consideraremos la expansión de este nodo, que se muestra en la siguiente figura:



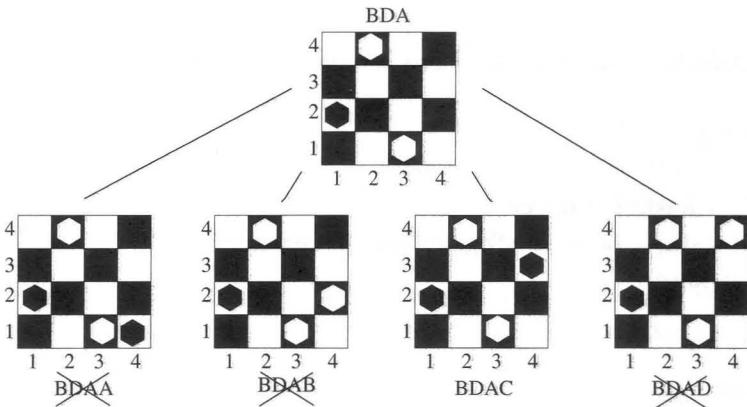
Pero en este caso no tenemos ninguna expansión completable. Esto nos lleva ha tener que **volver atrás** hasta el último nivel en el que dejamos algún nodo pendiente de explorar. En nuestro caso es el nodo *B* del nivel 1. La figura siguiente muestra las expansiones de este nodo:



Esta expansión nos deja un único nodo completable, *BD*, que expandimos de nuevo:



De esta expansión sólo resulta completable el nodo *BDA*, que se expande de nuevo:



En este caso hemos conseguido completar una solución, la correspondiente al nodo *BDAC*. Si sólo queremos una solución la exploración del grafo terminaría en este nodo.

Si se buscasen todas las soluciones, el algoritmo volvería hacia atrás para explorar los nodos pendientes.

Un posible algoritmo para encontrar todas las soluciones al problema de las reinas siguiendo el esquema de vuelta atrás sería el siguiente:

```
función Reinas(s:Vector[1..n] de entero, n,k: entero)
    s[k] ← 0
    mientras s[k] ≤ n hacer
        s[k] ← s[k] + 1
        si Completable(s,k) entonces
            si k = n entonces
                escribir(s)
            sino
                Reinas(s,n,k+1)
            fsi
        fsi
    fmientras
ffun
```

Este algoritmo recibe como parámetros un vector que almacena la solución s , el tamaño del tablero n , y la siguiente reina a colocar k . Se va probando a colocar la reina en las distintas filas, y si la solución parcial resultante de colocar esta reina es completable, se llama recursivamente a la función para que coloque las reinas restantes, a menos que ya se hayan colocado todas. En la llamada inicial el parámetro k toma el valor 1.

La función *completable* que aparece a continuación comprueba si una extensión de la solución parcial anterior es completable, es decir, no tiene reinas en la misma fila, ni en la misma diagonal.

```
fun Completable(s:vector[1..n] de entero, k: entero): booleano
    var
        i: entero
    fvar
    para i ← 1 hasta k-1 hacer
        si s[i] = s[k] ∨ (abs(s[i]-s[k]) = abs(i-k)) entonces
            dev falso
        fsi
    fpara
    dev cierto
ffun
```

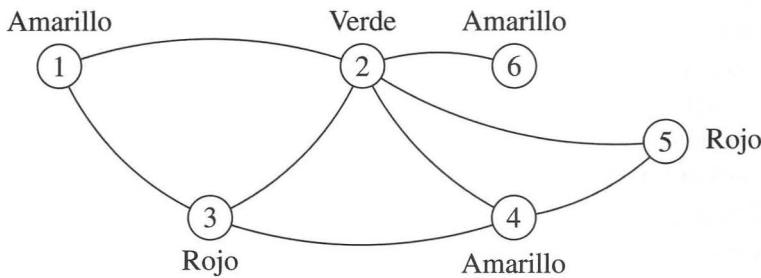
La función *abs* devuelve el valor absoluto.

Una cota superior al coste sería el tamaño del árbol completo, es decir $n!$. Al podar los nodos no completables, podemos asegurar que el coste estará por debajo de este valor.

6.2 Coloreado de grafos

En este problema se desea asignar un color de un conjunto de m colores a cada vértice de un grafo conexo, de manera que no haya dos vértices adyacentes que tengan el mismo color. La convención de usar colores tiene su origen en el problema de colorear mapas en un plano. Este problema se transforma en el coloreado de mapas asignando un vértice a cada país y un enlace entre los vértices que representan a países vecinos.

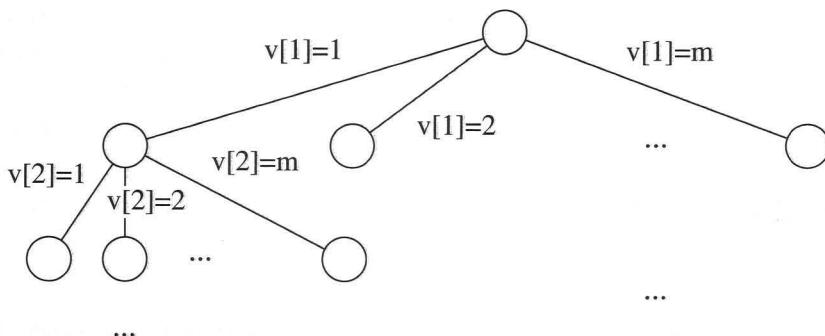
La siguiente figura muestra un ejemplo de grafo a cuyos vértices se han asignado colores de un conjunto de 3 colores distintos:



Se sabe que 4 colores son suficientes para colorear cualquier mapa en el plano, pero podemos necesitar menos, como ocurre en el ejemplo anterior. Nosotros vamos a considerar el número de colores una variable del problema.

Podemos numerar los vértices del grafo de 1 a n y representar así la solución por un vector en el que la posición i contiene el color asignado al vértice i . Así mismo podemos numerar los colores para representarlos por número naturales.

En el árbol de exploración de soluciones cada nivel corresponde a la asignación de color a un vértice, y cada nodo tiene m hijos que corresponden a las distintos colores que se puede asignar al vértice, como muestra la siguiente figura:



Un posible algoritmo que implementa el esquema de vuelta atrás para este problema es el siguiente:

```

tipo Grafo = matriz[1..N,1..N] de entero
tipo Vector = matriz[0..N] de entero
fun ColoreaGrafo (g:Grafo, m: entero, k: entero, v: Vector, exito:booleano)
    {v es un vector k-prometedor}
    v[k+1] ← 0
    exito ← falso
    mientras v[k+1] ≤ m ∧ ¬ exito hacer
        v[k+1] ← v[k+1] + 1
        si Completable (v) entonces
            si k = N entonces
                Procesar(v)
                exito ← cierto
            sino
                ColoreaGrafo(g,m,k+1,v,exito)
            fsi
        fsi
    fmiéntras
ffun

```

Representamos el grafo por la matriz de adyacencia en la que cada posición indica si hay un enlace entre los correspondientes nodos del grafo. La solución es un vector v en el que cada posición representa a un vértice del grafo. El valor que contiene cada posición es el color asignado al vértice. Los datos de entrada al algoritmo son el grafo, el número de colores disponibles, y la posición del vértice al que toca asignar color. Los

datos de salida son la solución (que también es de entrada) y un indicador de que se ha conseguido con éxito la asignación de colores. Cuando se procesa el último nodo del grafo ($k = N$) el algoritmo termina con éxito. Mientras eso no ocurre se van asignando colores al siguiente nodo del grafo, que está en la posición $k + 1$ del vector solución. Si un color no es válido se pasa a probar el siguiente, incrementando el valor asignado a $w[k + 1]$. La función *completable* se encarga de comprobar si un color es válido:

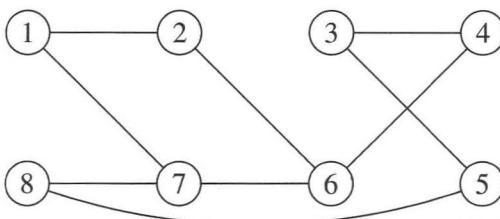
```
fun Completable(g:Grafo, v:Vector, k: entero): booleano
    i: entero
    para i  $\leftarrow 1$  hasta k-1 hacer
        si g[k,i] = 1  $\wedge$  v[k] = v[i] entonces
            dev falso
        fsi
    fpara
    dev cierto
ffun
```

Esta función recorre todos los nodos enlazados con el vértice k en el grafo y comprueba que tengan un color diferente al asignado al vértice k .

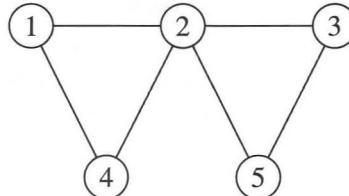
Para este algoritmo el espacio de búsqueda tiene la forma del árbol mostrado anteriormente, por lo que una cota al coste del algoritmo es $O(m^n)$.

6.3 Ciclos Hamiltonianos

Nos planteamos encontrar todos los ciclos Hamiltonianos de un grafo. Es decir, todos aquellos caminos que pasan por cada vértice una sola vez y terminan en el vértice inicial. Por ejemplo, si consideramos el siguiente grafo:



encontramos el ciclo Hamiltoniano 1-2-6-4-3-5-8-7-1. Pero en el siguiente grafo:



no encontramos ningún ciclo Hamiltoniano.

Cuando las aristas del grafo tienen asignado un valor que representa la distancia entre los vértices que unen, y buscamos el ciclo Hamiltoniano de longitud mínima, este problema se convierte en el conocido como el Viajante de Comercio que veremos en el capítulo siguiente.

Puesto que necesitamos explorar todos los posibles ciclos Hamiltonianos, y existe la posibilidad de tener que retroceder en la exploración del ciclo seguido porque nos lleve a un vértice ya visitado, el esquema de vuelta atrás es adecuado para resolver este problema.

Representamos el grafo por su matriz de adyacencia. Podemos representar la solución por un vector de vértices, de manera que el primer vértice del ciclo esté en la primera posición del vector, el siguiente en la segunda, etc. Para evitar soluciones iguales empezando en distinto vértice, fijamos un vértice de comienzo, por ejemplo el 1. El algoritmo tiene que retroceder a un punto anterior del árbol de búsqueda cuando llega a un nodo que no está conectado o cuando llega a un nodo que ya se ha incluido en el ciclo. Para facilitar la comprobación de los nodos incluidos usaremos un vector de booleanos, *incluidos*, que registre dichos nodos. Entonces el algoritmo puede tomar la siguiente forma:

```

tipo Grafo = matriz[1..N,1..N] de entero
tipo Vector = matriz[1..N] de entero
tipo VectorB = matriz[1..N] de booleano
fun CiclosHamiltoniano (g:Grafo, k: entero, v: Vector, incluidos: VectorB)
  var
    i: entero
  fvar
  para i = 2 hasta n hacer
    si g[v[k-1],i]  $\wedge$   $\neg$  incluidos[i] entonces
      v[k]  $\leftarrow$  i
      incluidos[i]  $\leftarrow$  cierto
      si k = n entonces
        fin si
      fin si
    fin si
  fin para

```

```

{ se comprueba que se cierra el ciclo}
si g[v[n],1] entonces
    PresentarSolución(v)
fsi
sino
    CiclosHamiltoniano(g,k+1,v, incluidos)
fsi
    incluidos[i] ← falso
fsi
fpara
ffun

```

El algoritmo busca todas las rutas accesibles desde todos los nodos excepto el primero. Si el nodo i considerado está conectado con el anterior de la ruta, y aún no se ha visitado se incluye en la ruta, marcándolo como incluido en el vector *incluidos*. Después se comprueba si se han completado los n nodos de la ruta. Si es así, se comprueba que la ruta sea circular, es decir, que el último nodo esté enlazado con el primero, en cuyo caso tendríamos una solución, que se presenta. Si aún no se ha completado la ruta se hace una llamada recursiva a *ciclosHamiltoniano* para que siga incluyendo nodos. El nodo marcado como *incluso* se desmarca para que en el siguiente ciclo del bucle *para* en el que se prueba otro valor para el vértice i , sea ese nuevo valor el que se marque, y no el anterior.

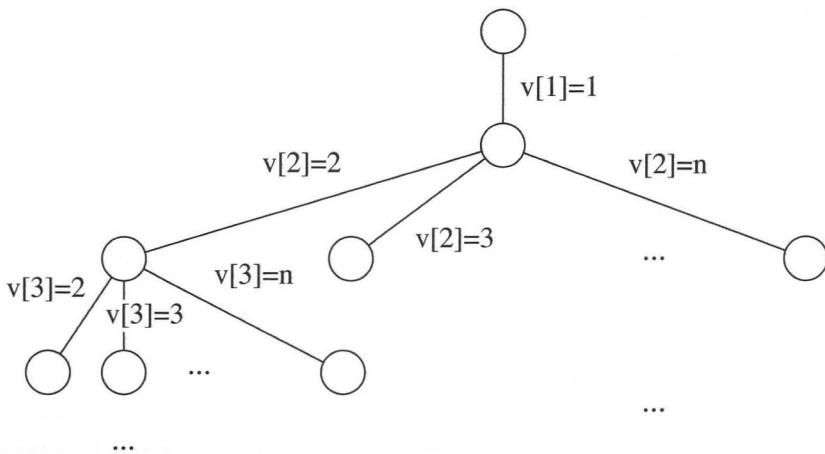
La llamada a la función recursiva que busca todos los ciclos Hamiltonianos requiere tener inicializados ciertos datos, como hace la siguiente función:

```

fun PresentarHamiltonianos(g)
    var
        v: Vector
        incluidos: VectorB
        i: entero
    fvar
        v[1] ← 1
        incluidos[1] ← cierto
    para i = 2 hasta n hacer
        incluidos[i] ← falso
    fpara
        CiclosHamiltoniano(g,2,v,incluidos)
ffun

```

El espacio de búsqueda para este problema tiene la siguiente forma:



Este árbol tiene n niveles y todos los nodos excepto la raíz, que tiene un hijo, tienen $n - 1$ hijos. Por tanto, una cota al coste del algoritmo es $O((n - 1)^n)$.

6.4 Subconjuntos de suma dada

Disponemos de un conjunto A de n números enteros sin repeticiones (tanto positivos como negativos) almacenados en una lista. Dados dos valores enteros m y C , siendo $m < n$ se desea resolver el problema de encontrar todos los subconjuntos de A compuestos por exactamente m elementos y tal que la suma de los valores de esos m elementos sea C .

Tenemos que recorrer un árbol de soluciones candidatas, pero siguiendo aquellas ramas que tengan opción de convertirse en una solución (k -prometedoras) por lo que el esquema de vuelta atrás es adecuado.

Dependiendo de la representación del vector solución y de la estrategia de ramificación (generación de hijos) podríamos considerar distintas aproximaciones para el esquema de vuelta atrás.

En una primera aproximación podría representarse una solución parcial como un vector de m enteros. Los descendientes de un nodo de nivel k serían todos los valores no tomados anteriormente. Así, tendríamos n opciones para el nivel 1 del árbol, en el nivel 2 tendríamos $n(n - 1)$ hijos, etc., hasta el nivel m .

Una cota superior al tiempo de ejecución de esta alternativa es el número de nodos del árbol, es decir las combinaciones de n elementos tomados de m en m , multiplicado por el número de posibles permutaciones, ya que éstas no se excluyen:

$$T(n, m) = \binom{n}{m} m! = \frac{n!}{(m-n)!m!} m! = \frac{n!}{(n-m)!} = n(n-1)\cdots(n-m+1)$$

Llegamos al mismo resultado calculando el número de hijos de cada nivel del árbol: el número de hijos de primer nivel es n , el numero de hijos del segundo nivel es $n(n - 1)$, hasta llegar al nivel $(n - m + 1)$, cuyo número de hijos es $n(n - 1) \cdots (n - m + 1)$. Podemos observar que si m se acerca a $n - 1$ (m es siempre menor que n) entonces $T(n) \in O(n!)$. Es decir, en el caso peor que es cuando m tiende a n , el número de nodos del árbol es $n!$.

Otra aproximación más adecuada, que es la que seguiremos aquí, consiste en representar la solución como un vector de n booleanos. De este modo, los descendientes de un nodo de nivel k serían las opciones de tomar o no el valor $k + 1$ del vector de entrada. Es decir siempre se tendrían dos opciones. Cuando m tiende a n , el número de nodos sería 2^n que es mejor que $n!$ Lo que ocurre es que esta segunda alternativa evita considerar permutaciones, es decir, tomar los mismos números pero en orden diferente.

Veamos como implementar esta segunda alternativa. El conjunto A de números se representa por un vector de enteros. Las soluciones candidatas las representaremos por un vector de booleanos en el que una posición con valor cierto indica que el número correspondiente a esa posición se incluye en la solución.

El algoritmo puede tomar la siguiente forma:

```

tipo Vector = matriz[1..N] de entero
tipo VectorB = matriz[1..N] de booleano
fun SubconjuntosSumaDada(datos: Vector, k: entero, v: VectorB,
                           sumandos: entero, suma: entero)
    var
        i: entero
    fvar
    si sumandos = m entonces
        si suma = C entonces
            PresentarSolución(v)
        fsi
    sino
        si k < n entonces
            v[k] ← falso
            SubconjuntosSumaDada(datos,k+1,v, sumandos, suma)
            v[k] ← cierto
            si suma + datos[k] ≤ C entonces
                suma ← suma + datos[k]
                sumandos ← sumandos + 1
                SubconjuntosSumaDada(datos,k+1,v, sumandos, suma)
            fsi
        fsi
    ffun
```

El algoritmo tiene como entrada el conjunto de números *datos*, la posición del número que se está considerando, un vector de booleanos *v*, en el que se marca los números seleccionados, el número de sumandos incluidos y la suma alcanzada hasta el momento. El algoritmo va considerando cada número del conjunto. Cuando se llega a utilizar un total de *m* números se comprueba si se ha alcanzado una solución, es decir, si la suma de los números seleccionados es *C*. Mientras no se tienen los *m* números se van generando dos alternativas para cada número del conjunto, que corresponden a no seleccionar y a seleccionar el siguiente número.

En la llamada inicial a *subconjuntosSumaDada*, todas las posiciones del vector *v* se inicializan a falso, el número de sumandos y la suma a 0, y *k* a 1.

Cada nivel el árbol se divide como máximo en dos ramas (si no se ha llegado a tener *m* posiciones con valor cierto), luego una cota superior es 2^n .

6.5 Reparto equitativo de activos

En este problema ayudaremos a dos socios que forman una sociedad comercial a dividirla. Cada uno de los *n* activos de la sociedad que hay que repartir tiene un valor entero positivo. Los socios quieren repartir dichos activos a medias y, para ello, quieren conocer todas las posibles formas que tienen de dividir el conjunto de activos en dos subconjuntos disjuntos, de forma que cada uno de ellos tenga el mismo valor entero.

Aunque a primera vista nos pudiera parecer que el problema puede resolverse mediante un esquema voraz, ordenando en primer lugar los *n* activos de mayor a menor, en realidad no es así. Supongamos que la función de selección consistiera en asignar al primer socio el mayor activo de los que quedan mientras no supere la mitad del total de valor de los activos. Entonces tendríamos un contraejemplo en la siguiente colección de activos, que ya se presentan ordenados de mayor a menor:

| | | | | | | | | | | |
|--------|----|---|---|---|---|---|---|---|---|----|
| Activo | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| Valor | 10 | 9 | 5 | 3 | 3 | 2 | 2 | 2 | 2 | 2 |

La suma de los valores de todos los activos de esta sociedad es 40. Así que a cada socio se le deben asignar activos por valor de 20. Si aplicamos el algoritmo voraz que hemos descrito se tendría el siguiente reparto de activos:

| Socio1 | Socio2 |
|--------|--------|
| 10 | 5 |
| 9 | 3 |
| | 3 |
| | 2 |
| | 2 |
| | 2 |
| | 2 |
| | 2 |
| | 2 |

Como podemos ver no es posible completar el reparto aplicando este algoritmo. Sin embargo, es fácil encontrar un reparto de este conjunto en dos subconjuntos de igual valor:

| Socio1 | Socio2 |
|--------|--------|
| 10 | 9 |
| 2 | 5 |
| 2 | 3 |
| 2 | 3 |
| 2 | |
| 2 | |

Por lo tanto el esquema voraz no es válido y recurrimos al esquema de vuelta atrás para explorar el grafo de soluciones. En este caso, el espacio de búsqueda es un árbol de grado 2 y altura $n + 1$. Cada nodo del i -ésimo nivel tiene dos hijos correspondientes a si el i -ésimo activo va a un socio o al otro.

Entre las estructuras de datos que vamos a utilizar están el vector con los valores de los activos de la sociedad y el vector en el que vamos construyendo la solución. Este último contendrá solo los valores 1 y 2 que indican si el activo de la posición i se ha asignado al socio 1 o al socio 2. Podemos plantear entonces el problema, aplicando el esquema general de vuelta atrás de la siguiente forma:

```

tipo Vector = matriz[0..N] de entero
fun DividirSociedad(x: Vector, suma1, suma2, sumaTotal, k: entero, v: Vector)
    {v es un vector k-prometedor}
    si k = N entonces
        si suma1 = suma2 entonces
            Procesar(v)
        fsi
    sino
        v[k+1] ← 1
        si Completable (x, suma1, sumaTotal, k+1) entonces
            suma1 ← suma1 + x[k+1]
        fsi
    
```

```

DividirSociedad(x, suma1, suma2, sumaTotal, k+1, v)
fsi
v[k+1] ← 2
si Completable (x, suma2, sumaTotal, k+1) entonces
    suma2 ← suma2 + x[k+1]
    DividirSociedad(x, suma1, suma2, sumaTotal, k+1, v)
fsi
fsi
ffun

```

Este algoritmo tiene como parámetros de entrada el vector de activos y las sumas de activos que ya se han asignado al socio 1, *suma1*, las que se han asignado al socio 2, *suma2*, y la suma total de los valores de los activos, *sumaTotal*. También tiene como parámetros la posición *k* del siguiente activo a asignar y el vector de asignaciones realizadas *v*. El algoritmo comprueba si se ha alcanzado una solución, es decir si ya se han asignado todos los activos, ($k = N$), y si se ha hecho de tal forma que la suma asignada a cada socio es la misma. Si aún quedan activos pendientes se construyen los vectores $k + 1$ -prometedores, que en este caso son dos y corresponden a asignar el siguiente activo $k + 1$ al socio 1 o al socio 2. En cada caso se comprueba si la asignación es válida mediante una llamada a la función *completable*. Esta función comprueba que la asignación del activo *k* a un determinado socio no lleva a que los valores asignados a ese socio superen la mitad del total.

```

fun Completable(x:Vector, sumaParcial, sumaTotal, k: entero): booleano
    si sumaParcial + x[k] ≤ sumaTotal div 2 entonces
        dev falso
    sino
        dev cierto
    fsi
ffun

```

La llamada a la función que reparte los activos requiere un procesamiento previo que consiste en calcular la suma total de los activos a dividir y en inicializar las variables de llamada. Antes de la llamada se comprueba que el valor total sea par para que sea posible el reparto en cantidades enteras.

```

fun ResolverSeparacionSocios (x:Vector)
    var
        i, suma1, suma2, sumaTotal: entero

```

v: Vector

fvar

sumaTotal \leftarrow 0

suma1 \leftarrow 0

suma2 \leftarrow 0

para i \leftarrow 1 hasta N **hacer**

sumaTotal \leftarrow sumaTotal + x[i]

fpara

si sumaTotal mod 2 = 0 **entonces**

DividirSociedad(x, v, 1, suma1, suma2, sumaTotal)

fsi

ffun

El coste viene dado por el número máximo de nodos del espacio de búsqueda. Como cada nodo del árbol se divide en dos que corresponden a asignar el siguiente activo al socio 1 o al socio 2, y como la profundidad del árbol viene dada por el número de activos n , entonces el coste está acotado por $O(2^n)$.

6.6 El robot en busca del tornillo

Este problema es un ejemplo de aplicación del esquema a la búsqueda de caminos en laberintos. Un robot se mueve en un edificio en busca de un tornillo. Se trata de diseñar un algoritmo que le ayude a encontrar el tornillo y a salir después del edificio. El edificio debe representarse como una matriz de entrada a la función, cuyas casillas contienen uno de los siguientes tres valores: L para “paso libre”, E para “paso estrecho” (no cabe el robot) y T para “tornillo”. El robot sale de la casilla (1,1) y debe encontrar la casilla ocupada por el tornillo. En cada punto, el robot puede tomar la dirección Norte, Sur, Este u Oeste siempre que no sea un paso demasiado estrecho. El algoritmo debe devolver la secuencia de casillas que componen el camino de regreso desde la casilla ocupada por el tornillo hasta la casilla (1,1). Supondremos que la distancia entre casillas adyacentes es siempre 1.

Como no es necesario encontrar el camino más corto, sino encontrar un camino lo antes posible, una búsqueda en profundidad resulta más adecuada que una búsqueda en anchura. Si el edificio fuera infinito entonces una búsqueda en profundidad no sería adecuada porque no garantiza que se pueda encontrar una solución. Nosotros suponemos que el edificio es finito. Al ir recorriendo casillas es posible que el robot llegue a una que no tenga salida, por lo que es necesario que el algoritmo disponga de un mecanismo para deshacer movimientos, lo que hace que el esquema de vuelta atrás sea el adecuado. Vamos a utilizar el esquema de vuelta atrás de forma que la búsqueda se detenga en la primera solución y devuelva la secuencia de casillas desde el tornillo hasta la salida.

También iremos registrando las casillas que ya se han explorado para evitar repeticiones. Para representar las casillas podemos utilizar un registro de dos enteros x e y , que nos indiquen una posición en el tablero. Para registrar los nodos explorados podemos utilizar una matriz del tamaño del edificio pero de valores booleanos. Vamos a utilizar una lista de casillas para almacenar la solución y otra para las extensiones del camino k -prometedoras. Contaremos con las siguientes funciones del tipo *lista*: *CrearLista*, *ListaVacia?*, *Añadir* y *Primero*. El algoritmo puede tomar entonces la siguiente forma:

```

tipo TEdificio = matriz[0..LARGO, 1..ANCHO] de caracter
tipo TEdificioB = matriz[0..LARGO, 1..ANCHO] de booleano
tipo TCasilla =
    registro
        x,y: entero
    fregistro
tipo TListaCasillas = Lista de TCasilla
fun BuscaTornillo (edificio: TEdificio, casilla: TCasilla,
    exploradas: TEdificioB, solucion: TListaCasillas, exito: booleano)
    exploradas[casilla.x, casilla.y] ← cierto
    si edificio[casilla.x, casilla.y] = T entonces
        solución ← CrearLista()
        solución ← Añadir(solución, casilla)
        exito ← cierto
    sino
        hijos ← Caminos(edificio, casilla)
        exito ← falso
        mientras ¬ exito ∧ ¬ ListaVacia?(hijos) hacer
            hijo ← Primero (hijos)
            si ¬ exploradas [hijo.x, hijo.y] entonces
                BuscaTornillo (edificio,hijo,exploradas,solución,exito)
            fsi
        fmientras
        si exito entonces
            solución ← Añadir(solución, casilla)
        fsi
    fsi
ffun

```

En el caso de encontrar el tornillo se detiene la exploración en profundidad y al deshacer las llamadas recursivas se van añadiendo a la solución las casillas que se han recorrido. Como se añaden al final de la lista, la primera será la del tornillo y la última la casilla (1,1), tal como queríamos. La función *caminos* comprueba que la casilla no es estrecha

y que no está fuera del edificio. En cada caso en que esto se cumple añade a la lista de hijos una nueva alternativa a explorar.

fun Caminos (edificio: TEdificio, casilla: TCasilla): TListaCasillas

var

 hijos: TListaCasillas

 casilla_aux: TCasilla

fvar

 hijos \leftarrow CrearLista()

si casilla.x+1 \leq LARGO **entonces**

si edificio[casilla.x+1,casilla.y] \neq E **entonces**

 casilla_aux.x \leftarrow casilla.x+1

 casilla_aux.y \leftarrow casilla.y

 hijos \leftarrow Añadir (solución, casilla_aux)

fsi

fsi

si casilla.x-1 \geq 1 **entonces**

si edificio[casilla.x-1,casilla.y] \neq E **entonces**

 casilla_aux.x \leftarrow casilla.x-1

 casilla_aux.y \leftarrow casilla.y

 hijos \leftarrow Añadir (solución, casilla_aux)

fsi

fsi

si casilla.y+1 \leq ANCHO **entonces**

si edificio[casilla.x,casilla.y+1] \neq E **entonces**

 casilla_aux.x \leftarrow casilla.x

 casilla_aux.y \leftarrow casilla.y+1

 hijos \leftarrow Añadir (solución, casilla_aux)

fsi

fsi

si casilla.y-1 \geq 1 **entonces**

si edificio[casilla.x,casilla.y-1] \neq E **entonces**

 casilla_aux.x \leftarrow casilla.x

 casilla_aux.y \leftarrow casilla.y-1

 hijos \leftarrow Añadir (solución, casilla_aux)

fsi

fsi

dev hijos

ffun

Suponemos “edificio” inicializado con la configuración del edificio y “exploradas” inicializado con todas las posiciones a falso.

El espacio de búsqueda del problema es un árbol en el que cada nodo da lugar como máximo a cuatro ramas correspondientes a las cuatro direcciones de búsqueda. El número de niveles del árbol es el número de casillas del tablero, n^2 . Luego una cota superior es 4^{n^2} .

6.7 Asignación de cursos en una escuela

En una nueva escuela se van a impartir n cursos. El equipo directivo les ha asignado n aulas y n profesores, que deben repartirse entre los cursos teniendo en cuenta que existen una serie de restricciones. No todas las aulas tienen capacidad para todos los cursos, pues estos cuentan con distinto número de alumnos. Por ello se dispone de una función booleana $válida(aula, curso)$ que indica si el aula tiene suficiente capacidad para el curso. Además, los profesores no están especializados en los temas de todos los cursos, por lo que también se dispone de una función $especialidad(prof, curso)$ que indica si el profesor está especializado en un determinado curso. Se pide encontrar un algoritmo que encuentre alguna forma de asignar a cada curso un aula y un profesor apropiados.

Podemos representar la solución como un vector de registros, cada uno de los cuales indica la asignación de un aula y un profesor a un curso. La posición del registro en el vector indica el curso de que se trata.

El algoritmo tiene que ir marcando las aulas y los profesores que ya se han asignado a un curso para que no se asignen a otros. Tras estas consideraciones podemos escribir un algoritmo para este problema de la siguiente forma:

```

tipo TCurso =
  registro
    aula: entero
    profesor: entero
  fregistro

tipo TEscuela = matriz[0..n] de TCurso
tipo TVectorB = matriz[0..n] de booleano
fun CursosEscuela (escuela: TEscuela, k: enteros, asigAula: TVectorB,
                     asigProf: TVectorB, exito: booleano)
var
  es_solucion: booleano
  aula: entero
  prof: entero
fvar
  aula ← 1
  mientras aula ≤ n ∧ ¬ exito hacer
    si ¬ asigAula[aula] ∧ válida(aula,k) entonces

```

```

prof ← 1
mientras prof ≤ n ∧ ¬ exito hacer
    si ¬ asigProf[prof] ∧ especialidad(prof,k) entonces
        escuela[k].aula ← aula
        escuela[k].prof ← prof
        asigAula[aula] ← cierto
        asigProf[prof] ← cierto
    si k = n entonces
        PresentarSolucion(escuela)
        exito ← cierto
    sino
        CursosEscuela(escuela,k+1,asigAula,asigProf,exito)
    fsi
    asigAula[aula] ← falso
    asigProf[prof] ← falso
fsi
    prof ← prof + 1
fmientras
fsi
    aula ← aula + 1
fmientras
ffun

```

Para un cierto curso k , el algoritmo va recorriendo las aulas mientras no consiga una asignación válida. Para cada aula que considera comprueba que no se le haya asignado ya a otro curso y que sea válida para el curso k que se está procesando. Si se cumplen estas condiciones pasa a buscar un profesor adecuado. De nuevo se recorren los profesores comprobando para cada uno de ellos que no esté ya asignado a otro curso y que la temática del curso k entre dentro de su especialidad. Si se cumplen estas condiciones se asignan al curso k el aula y profesor encontrados, marcándolos como asignados. Si no se ha completado la solución ($k = n$) se pasa a buscar asignaciones para los cursos restantes con una llamada recursiva a *cursosEscuela*. Antes de pasar a probar otras posibles asignaciones de aulas y profesores se desmarcan las asignaciones realizadas en el intento.

La llamada al algoritmo requiere asignar falso a *asigAula* y *asigProf* y llamar a la función que implementa el algoritmo con los parámetros adecuados:

```

fun HorarioValido(n:entero)
    var
        asigAula: TvectorB
        asigprof: TvectorB

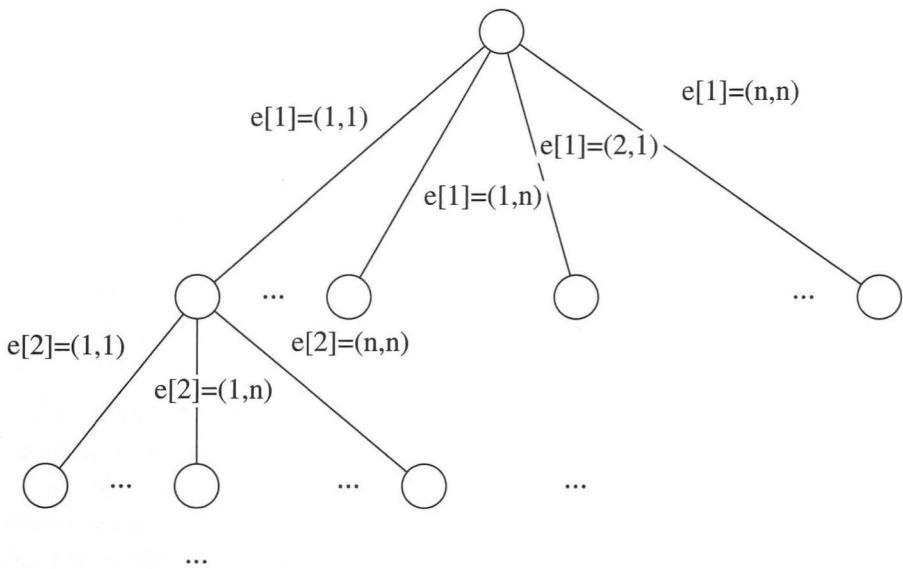
```

```

fvar
para i = 1 hasta n hacer
    asigAula  $\leftarrow$  falso
    asigProf  $\leftarrow$  falso
fpara
    exito  $\leftarrow$  falso
    CursosEscuela(escuela,1,asigAula,asigProf,exito)
ffun

```

El árbol de exploración del espacio de búsqueda tiene la siguiente forma:



es decir, cada nodo tiene n^2 hijos y el árbol tiene n niveles. Por tanto, una cota al coste es $O((n^2)^n) = n^{2n}$.

6.8 Ejercicios propuestos

1. Se considera un tablero de ajedrez $n \times n$ y un rey colocado en cierta posición (i, j) . Se pide un algoritmo que encuentre, si existe, una secuencia de $n^2 - 1$ movimientos del rey, de tal forma que se visiten todas las casillas del tablero una sola vez.
2. Se considera un tablero de ajedrez $n \times n$ y un caballo colocado en cierta posición (i, j) . Se pide un algoritmo que encuentre, si existe, una secuencia de $n^2 - 1$

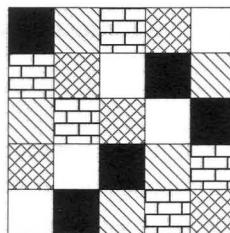
movimientos del caballo, de tal forma que se visiten todas las casillas del tablero una sola vez.

3. Se tiene un conjunto de n hombres y n mujeres, con distintos grados de preferencia entre ellos. Una pareja formada por un hombre h y una mujer m se considera *estable* si no se da ninguno de los dos hechos siguientes:

- (a) Existe otra mujer m' , que forma parte de la pareja (h', m') que es preferida por el hombre h sobre la mujer m , y además la mujer m' prefiere a h sobre h' .
- (b) Existe otra hombre h'' , que forma parte de la pareja (h'', m'') que es preferido por la mujer m sobre el hombre h , y además el hombre h'' prefiere a m sobre m'' .

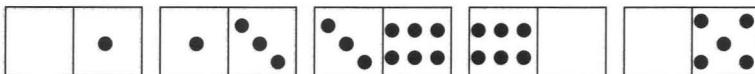
Se pide un algoritmo, que si existe, encuentre un emparejamiento de hombres y mujeres tal que todas las parejas sean estables. Las preferencias de los hombres y de las mujeres se representan por sendas matrices M y H , tales que la fila i de la matriz M tiene ordenadas de mayor a menor las preferencias de mujeres del hombre i , y la fila i de la matriz H tiene ordenadas de mayor a menor las preferencias de la mujer i .

4. Se pide un algoritmo de vuelta atrás para resolver el problema de la devolución de cambio, cuando se considera un conjunto de n tipos distintos de monedas y se quiere devolver una cantidad C utilizando un número mínimo de monedas.
5. Se tiene una serie de n cubos. Cada cubo tiene una letra distinta en cada una de sus caras. Dada un palabra de n letras, se pide un algoritmo que busque una colocación en secuencia de los n cubos de manera que se forme la palabra indicada.
6. Se considera una variante del problema de la sección 6.6 en el que se tiene un laberinto $n \times n$ representado por una matriz booleana $Lab[n \times n]$. Los movimientos que se pueden realizar vienen indicados por la matriz Lab . Si $Lab[i, j]$ es cierto entonces se puede pasar por la casilla (i, j) , pero si es falso no se puede. Se pide un algoritmo que encuentre algún camino, si existe, desde la casilla $(1, 1)$ a la casilla (n, n) . Se supone que se puede pasar por estas dos casillas.
7. Se considera un tablero $n \times n$ y un conjunto de n colores. Cuando a cada casilla se le asigna un color de los n disponibles de tal manera que no se repiten colores en ninguna fila ni en ninguna columna, el tablero se conoce como *cuadrado latino*. La siguiente figura muestra un ejemplo para $n = 5$:



Se pide diseñar un algoritmo que dados n colores presente todos los cuadrados latinos para un tablero $n \times n$.

8. Consideramos n tipos de sellos diferentes. Se tienen tres sellos de cada tipo, y en cada carta se pueden colocar un máximo de 5 sellos. Se pide un algoritmo que indique todas las formas posibles de franquear una carta si el orden de los sellos no importa.
9. Consideramos n letras diferentes y queremos diseñar un algoritmo de vuelta atrás que calcule todas las palabras (cualquier secuencia de letras) que se pueden formar tomando m letras ($m \leq n$) diferentes.
10. Se considera el juego del dominó con sus 28 fichas diferentes. Cada ficha está formada por dos secciones y en cada una de ellas está grabado un número de puntos entre 0 y 6. Las reglas del juego indican que las fichas se colocan formando una cadena de manera que dos fichas consecutivas tienen el mismo número de puntos en los segmentos que quedan contiguos. La siguiente figura muestra un ejemplo de una cadena de dominó:



Se pide diseñar un algoritmo que presente sin repeticiones las cadenas circulares correctas de las 28 fichas del juego.

6.9 Notas bibliográficas

Se pueden encontrar introducciones a este esquema en numerosos textos clásicos [BB06, CLRS01]. Este tema se trata así mismo en diversos libros de carácter práctico [Gue00, MOOMV03, GMCLMPGC03, GA97] que contienen colecciones de problemas resueltos con el esquema de vuelta atrás. En [Gue00] se pueden encontrar entre otros, el problema del rey de ajedrez, el de las parejas estables y el laberinto. En [MOOMV03] se resuelven entre otros los problemas del caballo de ajedrez, el de la formación de palabras con n letras dadas, el cuadrado latino, los n cubos y el franqueo de cartas. En [GA97] se describe el problema del dominó. En [GMCLMPGC03] se resuelven entre otros el problema de la devolución de cambio.

Capítulo 7

Ramificación y poda

Este capítulo describe el esquema de ramificación y poda que se emplea en aquellos problemas en los que el objetivo es la optimización de uno o más criterios en la solución alcanzada. Este esquema será menos eficiente que el voraz cuando ambos se pueden utilizar para realizar la optimización, por lo que su aplicación está indicada sólo cuando no se conoce un algoritmo voraz válido para el problema. Al igual que en el caso de esquema de vuelta atrás, se exploran todas las posibles alternativas que permitan llegar a una solución, pero en este caso, además de cortar la exploración de los caminos que no pueden alcanzar una solución, se evita la de aquellos que sólo pueden alcanzar una solución peor que otras, ya que el objetivo es la optimización. La presentación del esquema general se ilustra aplicándolo a una de las variantes del problema de la mochila, la mochila entera, en el que hay que seleccionar el conjunto de objetos más valiosos posible que quepan en una mochila de una determinada capacidad. Otros problemas presentados en el capítulo son: la asignación de tareas en una pastelería, el problema del viajante de comercio, la selección de cursos de formación, y la minimización de la distancia de edición. En todos los casos se identifican los elementos del esquema y se proporciona una cota superior al coste.

El estudio de este capítulo debe ser posterior al de los grafos y montículos, ya que se manejan en la implementación de este esquema. También es conveniente haber estudiado el capítulo dedicado a los algoritmos voraces para discernir cuando se aplica cada una de las técnicas, y el capítulo dedicado al esquema de vuelta atrás, que comparte ciertos aspectos del recorrido de un grafo con el de ramificación y poda.

7.1 Planteamiento general

Ramificación y poda también es un esquema para explorar un grafo dirigido implícito. En este caso lo que se busca es la solución óptima de un problema. En este esquema los nodos no se exploran siguiendo la secuencia en la que se han generado, como se hace en el esquema de vuelta atrás, sino que se utiliza la función que se quiere optimizar para

establecer preferencias entre los nodos pendientes de explorar. Ahora el recorrido se dirige por el nodo activo más prometedor, por lo que se requiere una cola de prioridad para la gestión de los nodos activos. Los montículos son particularmente adecuados para almacenar la cola de prioridad que utiliza este esquema para registrar los nodos que se han generado y aún no han sido explorados. Una vez seleccionado un nodo se procede con una fase de *ramificación*, en la que se generan distintas extensiones de la solución parcial correspondiente a dicho nodo. A continuación, este esquema no sólo *poda* aquellas ramas que no pueden llegar a una solución, sino también aquellas que no pueden mejorar el valor asignado a la solución por la función que se quiere optimizar. Por ello, en cada nodo calcularemos una cota optimista del posible valor de las soluciones que pueden construirse a partir de él. Si la cota indica que estas soluciones serán con seguridad peores que una solución factible ya encontrada (o que una cota pesimista de las soluciones posibles), entonces no tiene sentido seguir explorando esa parte del grafo y se poda. Es decir se realiza una *poda por factibilidad* y una *poda por cota*.

Podemos escribir el esquema general de ramificación y poda para un problema de minimización de la siguiente forma:

```
fun RamificacionYPoda (nodoRaiz, mejorSolucion: TNodo, cota: real)
    monticulo = CrearMonticuloVacio()
    cota = EstimacionPes(nodoRaiz)
    Insertar(nodoRaiz, monticulo)
    mientras ¬ MonticuloVacio?(monticulo) ∧
        EstimacionOpt(Primero(monticulo)) < cota hacer
            nodo ← ObtenerCima(monticulo)
            para cada hijo extensión válida de nodo hacer
                si solución(hijo) entonces
                    si coste(hijo) < cota entonces
                        cota ← coste(hijo)
                        mejorSolucion ← hijo
                    fsi
                sino
                    si EstimacionOpt(hijo) < cota entonces
                        Insertar(hijo, monticulo)
                    fsi
                fsi
            fpara
        fmientras
ffun
```

El algoritmo va acumulando en un montículo los elementos que una vez completados pueden ser soluciones al problema. En el montículo se mantienen ordenados por una

estimación optimista (*EstimaciónOpt*) o a la baja (si consideramos una minimización) del valor que puede alcanzar. El algoritmo usa una cota para podar aquellos nodos para los que sea posible estimar que en el mejor de los casos no van a poder dar lugar a una solución mejor que la asociada a la cota. Esta cota es el valor de la mejor solución alcanzada hasta el momento. Pero el proceso antes de construir la primera solución y disponer de una cota puede ser largo. Por ello, para poder empezar a podar desde el principio buscamos una cota pesimista (*EstimaciónPes*), que representa el valor que tendría la solución en un caso cualquiera. Las soluciones que se consideran deben ser al menos mejor que este valor.

Al igual que ocurre en el esquema de vuelta atrás, el coste del problema depende del tamaño del árbol de posibles soluciones, por lo que en general daremos una cota superior al coste calculada en función de dicho tamaño.

Un ejemplo ilustrativo del esquema puede ser el conocido problema de la mochila entera. En este problema se tienen n objetos, o_1, o_2, \dots, o_n , cada uno de ellos con un peso p_1, p_2, \dots, p_n y un valor v_1, v_2, \dots, v_n , asociados. Se dispone sólo de una mochila que soporta un peso máximo P . El objetivo de este problema es hacer una selección de objetos de forma que se maximice el valor del contenido de la mochila, y cuya suma de pesos no sobrepase su capacidad. Los objetos son indivisibles, es decir, cada uno de ellos o se toma entero o se deja.

Este problema no puede ser resuelto por un algoritmo voraz, como ocurre en el caso en que los objetos se pueden dividir. Como tenemos que maximizar el valor del contenido estamos ante un problema de optimización, para el que el esquema de ramificación y poda es adecuado.

Podemos representar la solución al problema como un vector de booleanos que nos indique con un 1 (cierto) que el objeto de la posición i se ha incluido en la mochila, y con un 0 (falso) que no se ha incluido:

| | | | |
|-------|-------|---------|-------|
| o_1 | o_2 | \dots | o_n |
| 0 | 1 | \dots | 1 |

Para plantear el algoritmo necesitamos una estimación optimista, que en este caso es una cota superior al valor alcanzable ya que se trata de maximizar un valor. Podemos obtener un cota superior como el valor al que podríamos llegar si los objetos fueran divisibles. Este valor se calcula mediante un algoritmo voraz que va seleccionando objetos por su valor específico de mayor a menor. Recordamos que el valor específico de un objeto se calcula como su valor por unidad de peso, es decir $e_i = v_i / p_i$.

Necesitamos también una estimación pesimista del valor que como mínimo tendrá una solución. Una posibilidad es tomar como estimación la suma de los valores de los objetos que ya están metidos en la mochila. Podemos refinar la estimación si a ese valor se le suman los valores de los objetos que se puedan incorporar sin sobrepasar el peso siguiendo el orden dado por el valor específico de los objetos.

Como en el esquema general, utilizamos un montículo, que en este caso de máximos y está ordenado por valor específico, para almacenar los nodos pendientes de explorar.

Cada nodo registra la solución parcial que le corresponde, es decir, los objetos que ya se han introducido en la mochila. Almacena también la etapa de búsqueda que le corresponde, es decir la posición del objeto que toca considerar. Para mejorar la eficiencia, registramos también el peso, el valor total de los objetos introducidos en la mochila y la estimación optimista correspondiente al nodo, que determina su posición en el montículo.

tipo TVectorB = matriz[0..n] de booleano

tipo TVectorR = matriz[0..n] de real

tipo TNodo = registro

 moch: TVectorB

 k: entero

 pesoT: real

 valorT: real

 estOpt: real

fregistro

fun Mochila (pesos, valores: TVectorR, P: real, moch: TVectorB, valor: real)

var

 monticulo: TMonticulo

 nodo,hijo: TNodo

 cota,estPes: real

fvar

 monticulo \leftarrow CrearMonticuloVacio()

 valor \leftarrow 0

 {Construimos el primer nodo}

 nodo.moch \leftarrow moch

 nodo.k \leftarrow 0

 nodo.pesoT \leftarrow 0

 nodo.valorT \leftarrow 0

 nodo.estOpt \leftarrow EstimacionOpt(pesos, valores, P, nodo.k, nodo.pesoT, nodo.valorT)

 Insertar(nodo, monticulo)

 cota \leftarrow EstimacionPes(pesos, valores, P, nodo.k, nodo.PesoT, nodo.valorT)

mientras \neg MonticuloVacio?(monticulo) \wedge

 EstimacionOpt(Primero(monticulo)) $>$ cota **hacer**

 nodo \leftarrow ObtenerCima(monticulo)

 {se generan las extensiones válidas de nodo}

 {se mete el objeto en la mochila}

 hijo.k \leftarrow nodo.k + 1

 hijo.moch \leftarrow nodo.moch

si nodo.pesoT + pesos[hijo.k] \leq P **entonces**

 hijo.moch[hijo.k] \leftarrow cierto

 hijo.pesoT \leftarrow nodo.pesoT + pesos[hijo.k]

```

hijo.valorT ← nodo.valorT + valores[hijo.k]
hijo.estOpt ← nodo.estOpt
si hijo.k = n entonces
    si valor < hijo.valorT entonces
        moch ← hijo.moch
        valor ← hijo.valorT
        cota ← valor
    fsi
fsi
sino {la solución no está completa}
    Insertar(hijo, monticulo)
fsi
fsi
{no se mete el objeto en la mochila}
hijo.estOpt ← EstimacionOpt(pesos, valores, P, hijo.k,
    nodo.PesoT, nodo.valorT)
si hijo.estOpt ≥ cota entonces
    hijo.moch[hijo.k] ← falso
    hijo.pesoT ← nodo.pesoT
    hijo.valorT ← nodo.valorT
    si hijo.k = n entonces
        si valor < hijo.valorT entonces
            moch ← hijo.moch
            valor ← hijo.valorT
            cota ← valor
        fsi
fsi
sino {la solución no está completa}
    Insertar(hijo, monticulo)
    estPes ← EstimacionPes(pesos, valores, P, hijo.k,
        hijo.PesoT, hijo.valorT)
    si cota < estPes entonces
        cota ← estPes
    fsi
fsi
fmiendras
ffun

```

El algoritmo comienza construyendo el nodo inicial de la exploración. Suponemos que en la llamada inicial el vector *moch*, que va a contener los objetos seleccionados para

estar en la mochila, está inicializado a ceros (falso), ya que la mochila está vacía. Este nodo, incluyendo la estimación optimista del valor que puede llegar a alcanzar, *estOpt*, se introduce en el montículo. Se calcula también a partir de este nodo la cota inicial que sirve para podar aquellos nodos que no tengan opciones de superar este valor con el contenido de la mochila. Después, mientras queden nodos en el montículo, y siempre que la estimación de su valor sea mejor que la cota, se van sacando del montículo y generando los hijos correspondientes a las posibles extensiones que se tienen al considerar el siguiente objeto de la lista. En este caso las extensiones corresponden a incluir el siguiente objeto en la mochila o no incluirlo. En el nodo hijo correspondiente a incluir el objeto no es necesario calcular las estimaciones porque éstas ya suponen el objeto incluido, y por tanto no cambian. Lo que si se comprueba es que quede capacidad libre en la mochila para el objeto. Si es así se genera el nodo hijo y se comprueba si se ha llegado a completar una solución ($k = n$). Si se ha completado una solución, y ésta es mejor que una posible solución anterior, se almacena como mejor solución hasta el momento. Si no se ha completado una solución se añade el nodo al montículo. El otro hijo corresponde a no incluir el objeto en la mochila. En este caso sí que es necesario calcular la estimación optimista, para lo que se llama a la función correspondiente con los datos del nodo padre en el que no se había incluido el objeto. Esta extensión sólo se considera si la estimación optimista supera la cota mínima. Si se cumple esta condición se genera el hijo y de nuevo se comprueba si se ha completado una solución, procediendo análogamente al hijo anterior.

Para calcular las estimaciones es necesario que los vectores *pesos* y *valores* estén previamente ordenados por valor específico de los objetos.

La estimación optimista toma inicialmente el valor total de los objetos que ya se han incluido en la mochila, *valorT*. Después va añadiendo objetos por orden decreciente de valores específicos. Si el siguiente objeto cabe entero en la mochila se mete. Si no cabe se calcula el valor de la parte que cabría si los objetos fueran divisibles y se añade al valor total.

```

fun EstimacionOpt (pesos, valores: TVectorR, P: real, k: entero,
                    pesoT: real, valorT: real): real
var
    capacidad, estimacion: real
    i: entero
fvar
    capacidad ← P - pesoT
    estimacion ← valorT
    i ← k + 1
mientras i ≤ n ∧ capacidad ≥ 0 hacer
    si pesos[i] ≤ capacidad entonces
        estimacion ← estimacion + valor[i]

```

```

capacidad ← capacidad - pesos[i]
sino
  estimacion ← estimacion + (capacidad / pesos[i])*valor[i]
  capacidad ← 0
fsi
  i ← i + 1
fmientras
dev estimacion
ffun

```

La estimación pesimista también parte del valor de los objetos que están en la mochila. También va considerando objetos en orden decreciente de sus valores específicos, pero en este caso sólo se introducen en la mochila si caben enteros.

```

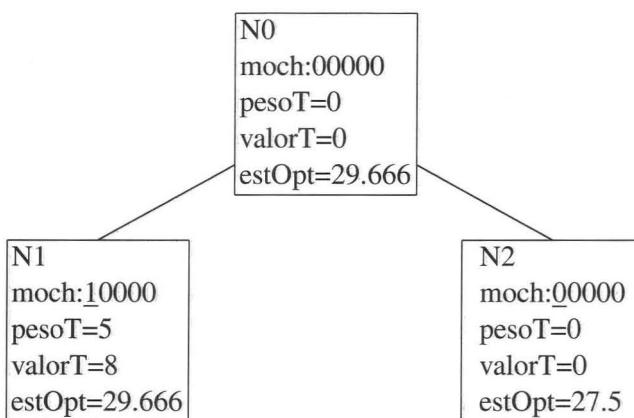
fun EstimacionPes (pesos, valores: TVectorR, P: real, k: entero,
                    pesoT: real, valorT: real): real
var
  capacidad, cota: real
  i: entero
fvar
  capacidad ← P - pesoT
  cota ← valorT
  i ← k + 1
mientras i ≤ n ∧ capacidad ≥ 0 hacer
  si pesos[i] ≤ capacidad entonces
    cota ← cota + valor[i]
    capacidad ← capacidad - pesos[i]
  fsi
  i ← i + 1
fmientras
dev cota
ffun

```

Supongamos que tenemos la siguiente instancia del problema para una mochila que soporta un peso máximo $P = 20$:

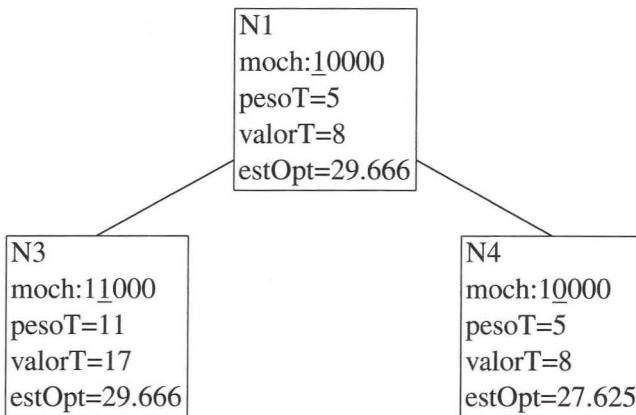
| Objeto | Peso | Valor | Valor Esp. |
|--------|------|-------|------------|
| o_1 | 5 | 8 | 1.6 |
| o_2 | 6 | 9 | 1.5 |
| o_3 | 4 | 6 | 1.5 |
| o_4 | 6 | 8 | 1.333 |
| o_5 | 8 | 9 | 1.125 |

La tabla muestra para cada objeto su peso, su valor y su valor específico (valor / peso). Los objetos ya están ordenados en la tabla de mayor a menor por valor específico. Mostramos a continuación parte de la exploración de espacio de soluciones para este problema.

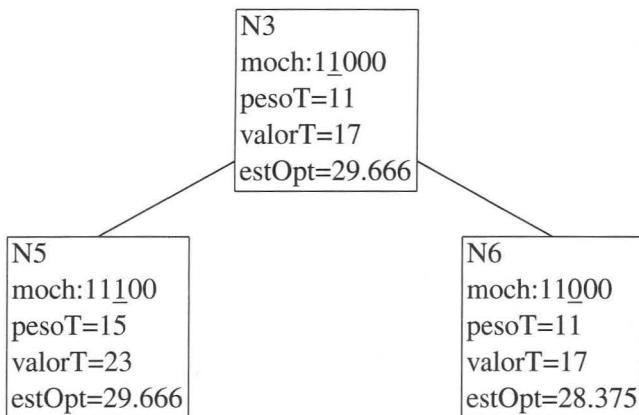


En cada nodo se representa el contenido en curso de la mochila por un vector de ceros y unos. La posición del objeto cuya inclusión se considera en el nodo es el dígito subrayado. Cada nodo presenta también el valor total y el peso total de los objetos que se han incluido en la mochila, así como una estimación optimista del valor que podría alcanzar una solución derivada de ese nodo. La estimación pesimista del nodo inicial produce una cota de valor 23, que en este caso coincidiría con el valor que produciría un algoritmo voraz. Este valor se utiliza inicialmente para podar aquellos nodos cuya estimación optimista no supere la cota. El valor de la cota se va actualizando a medida que se producen los nodos y las soluciones.

De los dos nodos N1 y N2 en los que se ha ramificado el nodo N0, la ordenación que nos proporciona el montículo, de mayor a menor estimación, nos llevaría a explorar antes el nodo N1:



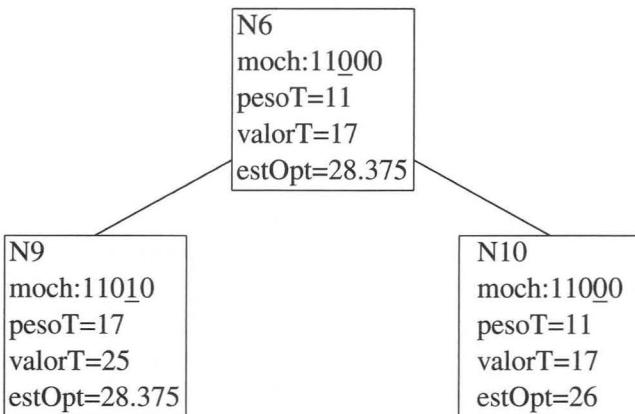
Vemos que la cota no poda ninguno de estos nodos porque sus estimaciones son superiores. Tampoco se actualiza su valor porque las cotas pesimistas calculadas para estos nodos son inferiores. A continuación el algoritmo consideraría el nodo N3 que es el que tiene la mayor estimación:



Al calcular la cota pesimista asociada a N6 se encuentra un valor de 25, por lo que se actualiza la cota de poda de 23 a 25.

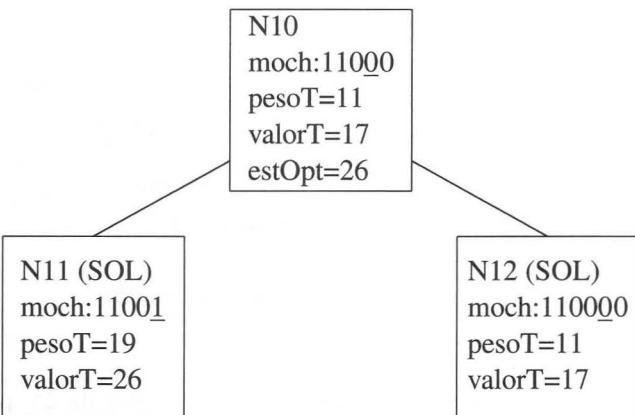
A continuación se exploraría el nodo N5, que sólo se ramificaría en otro nodo N7 correspondiente a la secuencia 11100, ya que el otro nodo no sería válido por sobrepasar el peso de la mochila. La estimación optimista para este nodo N7 es 30.875, por lo que sería el siguiente a explorar. De nuevo N7 sólo se ramifica en un nodo N8 por las limitaciones de la mochila. N8, al que corresponde la secuencia 11100 es la primera solución, que se almacena como la mejor hasta el momento. El valor de esta solución es 23, por lo que la cota de poda no se actualiza.

El siguiente nodo a explorar de los generados es N6, que produce los nodos N9 y N10:



El siguiente nodo a explorar sería N9 que sólo ramifica en un nodo que es 11010, que es una nueva solución de valor 25. Esta solución actualiza a la mejor solución encontrada hasta el momento. Además el nodo N10 da lugar a una estimación pesimista de valor 26, que actualiza el valor de la cota de poda.

El siguiente nodo que se explora es N10:



Este nodo lleva a dos nuevas soluciones, la primera de las cuales es la mejor, como se comprueba al final del algoritmo. La cota de poda, 26, que se tiene en este momento, y que coincide con el valor de la mejor solución alcanzada, permite podar varios de los nodos del árbol de soluciones.

En cuanto al coste, una cota superior del mismo es una estimación del número de nodos del árbol de exploración. Como cada nodo ramifica como máximo en 2 nodos y hay n niveles, el coste del algoritmo está en $O(2^n)$.

7.2 Asignación de tareas: Pastelería

Una pastelería tiene empleados a n pasteleros, que aunque son capaces de hacer cualquiera de los m tipos de pasteles distintos que ofrece la pastelería, tienen distinta destreza, y por tanto rendimiento, en la preparación de cada uno de ellos. Se desea asignar los próximos n pedidos, uno a cada pastelero, minimizando el coste total de la preparación de todos los pasteles. Para ello se conoce de antemano la tabla de costes $C[1..n, 1..m]$ en la que el valor c_{ij} corresponde al coste de que el pastelero i realice el pastel j , y los tipos de pasteles correspondientes a los pedidos, $pedidos[1..n]$.

Se trata de un problema típico de asignación de tareas, en los que hay que asignar una serie de tareas o recursos de forma que se optimice algún criterio. Al tratarse de una optimización, el esquema voraz podría ser apropiado si encontrásemos una función de selección adecuada. Sin embargo, es fácil comprobar que tal función no existe. Supongamos por ejemplo que tenemos 5 pasteleros y 3 tipos de pasteles. La siguiente tabla muestra el tiempo que le lleva a cada pastelero la realización de cada uno de los tres pasteles.

| | Pastel | | |
|-----------|--------|---|---|
| Pastelero | 1 | 2 | 3 |
| 1 | 2 | 5 | 3 |
| 2 | 5 | 3 | 2 |
| 3 | 6 | 4 | 9 |
| 4 | 6 | 3 | 8 |
| 5 | 7 | 5 | 8 |

Supongamos también que se tienen 5 pedidos de tipos [11321], es decir, se han pedido dos pasteles de tipo 1, uno de tipo 3, uno de tipo 2 y otro de tipo 1. Entonces un algoritmo voraz que tomase para cada pastel del pedido el pastelero que prepara con menos coste el tipo de pastel correspondiente, si está libre, y sino el siguiente mejor, asignaría los pasteleros: 12435. Esta asignación supone un coste total de 29. Pero una asignación como 13245 tiene un coste de 20. Aplicamos por tanto el esquema de ramificación y poda.

Para explorar el espacio de búsqueda utilizaremos nodos que almacenen un vector de n posiciones en el que cada posición i indica el índice del pastelero al que se ha asignado el pedido i . Los nodos también almacenan la posición del último pedido asignado y los pasteleros que aún no tienen asignado ningún pedido, además de la estimación optimista del coste de la solución a la que puede llegar el nodo. Utilizaremos también un montículo de mínimos para almacenar los nodos de forma que podamos ir tomando primero los de mejor estimación, en este caso los de menor coste total estimado.

tipo TVectorB = matriz[0..n] de booleano

tipo TVector = matriz[0..n] de entero

tipo TNodo = registro

pasteleros: TVector
asignados: TVectorB
k: entero
costeT: real
estOpt: real

fregistro

El campo *pasteleros* almacena en la posición *i* el pastelero asignado al pedido *i*. El campo *asignados* indica si el pastelero de la posición *i* ha sido ya asignado a la preparación de algún pastel. El campo *k* indica el último pedido asignado, *costeT* el coste total de las asignaciones ya hechas, y *estOpt* una optimización optimista, a la baja, del coste que puede alcanzar una solución a la que se llegue desde el nodo.

Necesitamos definir las estimaciones pesimista y optimista que utilizaremos para podar los nodos sin oportunidad de mejorar el resultado final.

Podemos definir una cota pesimista de una solución parcial sumando el coste de los pasteles ya asignados con el coste máximo que pueden tener los pendientes de asignar. Esta cota se actualizará cuando se alcancen soluciones y cuando se produzcan nuevos nodos.

Para la estimación optimista, que en este caso es mínima, podemos sumar el coste de la tareas ya asignadas con el coste mínimo que tienen las tareas pendientes de asignar.

tipo TTabla = matriz[1..n,1..n] de entero

**fun AsignaPasteleros (costes: TTabla, pedido: TVector,
pasteleros: TVector, costeT: entero)**

var

monticulo: TMonticulo
nodo,hijo: TNodo
cota,estPes: real

fvar

monticulo \leftarrow CrearMonticuloVacio()

costeT \leftarrow 0

{Construimos el primer nodo}

nodo.pasteleros \leftarrow pasteleros

para i \leftarrow 0 hasta n hacer

nodo.asignados[i] \leftarrow falso

fpara

nodo.k \leftarrow 0

nodo.costeT \leftarrow 0

nodo.estOpt \leftarrow EstimacionOpt(costes,pedido,nodo.k,nodo.costeT)

Insertar(nodo, monticulo)

```

cota ← EstimacionPes(costes, pedido, nodo.k, nodo.costeT)
mientras ¬ MonticuloVacio?(monticulo) ∧
    EstimacionOpt(Primero(monticulo)) < cota hacer
    nodo ← ObtenerCima(monticulo)
    { se generan las extensiones válidas del nodo}
    { para cada pastelero no asignado se crea un nodo}
    hijo.k ← nodo.k + 1
    hijo.pasteleros ← nodo.pasteleros
    hijo.asignados ← nodo.asignados
    para i ← 0 hasta n hacer
        si ¬ hijo.asignado[i] entonces
            hijo.pasteleros[hijo.k] ← i
            hijo.asignados[i] ← cierto
            hijo.costeT ← nodo.costeT + coste[i,pedido[hijo.k]]
        si hijo.k = n entonces
            si costeT > hijo.costeT entonces
                pasteleros ← hijo.pasteleros
                costeT ← costeT.valorT
                cota ← costeT
            fsi
        sino { la solución no está completa}
            hijo.estOpt ← EstimacionOpt(costes, pedido, hijo.k, nodo.costeT)
            Insertar(hijo, monticulo)
            estPes ← EstimacionPes(costes, pedido, hijo.k, hijo.costeT)
            si cota > estPes entonces
                cota ← estPes
            fsi
        fsi
        hijo.asignados[i] ← falso { se desmarca}
    fpara
fmientras
ffun

```

El algoritmo construye un nodo inicial sin ningún pastelero asignado a la preparación de un pastel, con k igual a 0, y con $costeT$ también 0. Este nodo inicializa el montículo. Después tenemos un bucle que no termina hasta que se vacía el montículo o hasta que el primer candidato no puede ser mejor que la solución actual con lo que no puede haber ninguna solución mejor. En cada iteración del bucle se comprueba que la estimación del coste de la solución a la que puede llegar el nodo no supere la cota. Si es así se construyen nodos con las extensiones posibles del nodo, que corresponden a los distintos pasteleros que se pueden asignar al siguiente pedido considerado (los que no estén ya asignados). Cuando en uno de esos nuevos nodos se comprueba que se ha alcanzado una solución, está sustituye a la anterior, si la hubiese, si es mejor que ella.

Las funciones que calculan las estimaciones pueden tomar la siguiente forma:

```
fun EstimacionOpt (costes: TTabla, pedido: TVector, k: entero, costeT: real): real
  var
    estimacion, menorC: real
    i,j: entero
  fvar
    estimacion ← costeT
    para i ← k+1 hasta n hacer
      menorC ← costes[1,pedido[i]]
      para j ← 2 hasta n hacer
        si menorC > costes[j,pedido[i]] entonces
          menorC ← costes[j,pedido[i]]
        fsi
      fpara
      estimacion ← estimacion + menorC
    fpara
    dev estimacion
ffun
```

Para calcular la estimación optimista se parte del coste de los pedidos ya asignados. Para cada uno de los pedidos que aún están por asignar se busca el menor coste posible, y se va sumando a la estimación.

La función que calcula la estimación pesimista es análoga, pero en este caso lo que se suma para cada pedido pendiente de asignar es el mayor coste posible.

```
fun EstimacionPes (costes: TTabla, pedido: TVector, k: entero, costeT: real): real
  var
    estimacion, mayorC: real
    i,j: entero
  fvar
```

```

estimacion ← costeT
para i ← k+1 hasta n hacer
    mayorC ← costes[1,pedido[i]]
    para j ← 2 hasta n hacer
        si mayorC < costes[j,pedido[i]] entonces
            mayorC ← costes[j,pedido[i]]
        fsi
    fpara
    estimacion ← estimacion + mayorC
fpara
dev estimacion
ffun

```

Una cota superior al coste del algoritmo es una estimación del tamaño del árbol. Cada nodo del árbol se expande en las $n - k$ asignaciones de pedidos pendiente y tiene n niveles, con lo que una cota al tamaño del árbol es $O(n!)$.

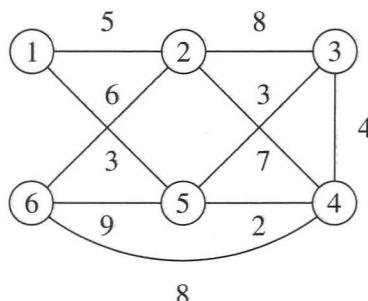
7.3 El viajante de comercio

En el capítulo dedicado al esquema de vuelta atrás vimos un algoritmo de búsqueda de los ciclos Hamiltonianos en un grafo, es decir, los caminos que pasan una sola vez por cada vértice y terminan en el vértice inicial. Consideraremos ahora que las aristas del grafo tienen asignado un valor y que buscamos el ciclo Hamiltoniano que partiendo de un nodo específico hace el recorrido de coste mínimo. Este problema se conoce como el “Viajante de comercio”, ya que se plantea como el problema de un viajante de comercio, que partiendo de la ciudad origen, tiene que visitar todas las ciudades de su zona una y sólo una vez y volver a la ciudad de origen, minimizando el coste del recorrido.

Estamos ante un problema de optimización para el que no podemos encontrar una solución voraz, y aplicamos por tanto el esquema de ramificación y poda.

Podemos representar la solución como un vector en el que el contenido de la posición i indica la ciudad por la que se pasa en el orden i del recorrido. El grafo lo representamos por la matriz de adyacencia que en la posición (i,j) indica el valor del enlace entre el vértice i y el j . Cuando no existe el enlace asignamos a la correspondiente posición de la matriz un valor especial que representamos por ∞ .

Por ejemplo, si consideramos el siguiente grafo:



Lo representaríamos por la matriz:

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|----------|----------|----------|----------|----------|----------|
| 1 | ∞ | 5 | ∞ | ∞ | 3 | ∞ |
| 2 | 5 | ∞ | 8 | 7 | ∞ | 6 |
| 3 | ∞ | 8 | ∞ | 4 | 3 | ∞ |
| 4 | ∞ | 3 | 4 | ∞ | 2 | 8 |
| 5 | 3 | ∞ | 3 | 2 | ∞ | 9 |
| 6 | ∞ | 6 | ∞ | 8 | 9 | ∞ |

Utilizaremos un montículo para procesar los nodos candidatos empezando por los de menor coste estimado. En cada nodo representaremos los vértices ya asignados en un vector *ruta*. Anotaremos también los vértices que ya han sido asignados para facilitar la exploración. Registraremos asimismo, la etapa de la ruta que vamos explorando, el coste de las asignaciones ya realizadas y una estimación optimista del coste de la solución que se podría alcanzar desde el nodo.

tipo TVectorB = matriz[0..n] de booleano

tipo TVector = matriz[0..n] de entero

tipo TNodo = registro

 ruta: TVector

 asignados: TVectorB

 k: entero

 costeT: real

 estOpt: real

fregistro

Consideramos ahora qué criterios podemos utilizar para calcular las estimaciones pessimista y optimista que utilizará el algoritmo para realizar las podas. La estimación optimista la podemos calcular sumando al coste de los trayectos correspondiente a los vértices ya asignados, el coste mínimo de cualquier trayecto. Es decir, buscamos el menor valor de cualquier arista.

Al considerar la estimación pesimista, hay que tener en cuenta que este problema puede no tener solución. Por lo tanto, no podemos podar una solución candidata porque su estimación optimista sea menor que una cierta cota a menos que ya hayamos encontrado una solución y por tanto estemos seguros de que el problema tiene solución.

Tras estas consideraciones podemos plantear el siguiente algoritmo de ramificación y poda, que busca ciclos Hamiltonianos mínimos que empiezan y terminan en el nodo 1 del grafo:

```

tipo TGrafo = matriz[1..n,1..n] de entero
tipo TVector = matriz[1..n] de entero
fun Viajante (grafo: TGrafo, ruta: TVector, costeT: entero)
    var
        monticulo: TMonticulo
        nodo,hijo: TNodo
        cota, estPes, verticeAnt, minArista: entero
    fvar
        monticulo ← CrearMonticuloVacio()
        minArista ← menorArista(grafo)
        costeT ← ∞
        {Construimos el primer nodo}
        nodo.ruta ← ruta
        nodo.asignados[1] ← cierto
        para i ← 2 hasta n hacer
            nodo.asignados[i] ← falso
        fpara
        nodo.k ← 1
        nodo.costeT ← 0
        nodo.estOpt ← EstimacionOpt(grafo,minArista,nodo.k,nodo.costeT)
        Insertar(nodo, monticulo)
        cota ← ∞
        mientras ¬ MonticuloVacio?(monticulo) ∧
            EstimacionOpt(Primero(monticulo)) < cota hacer
            nodo ← ObtenerCima(monticulo)
            {se generan las extensiones válidas de nodo}
            verticeAnt ← nodo.ruta[nodo.k]
            hijo.k ← nodo.k + 1
            hijo.ruta ← nodo.ruta
            hijo.asignados ← nodo.asignados
            para i ← 2 hasta n hacer
                si ¬ hijo.asignado[i] ∧ grafo[verticeAnt,i] ≠ ∞ entonces
                    hijo.ruta[hijo.k] ← i

```

```

hijo.asignados[i] ← cierto
hijo.costeT ← nodo.costeT + grafo[verticeAnt,i]
si hijo.k = n entonces
    si grafo[i,1] ≠ ∞ entonces
        si costeT > hijo.costeT entonces
            ruta ← hijo.ruta
            costeT ← hijo.costeT
            cota ← costeT
        fsi
    fsi
sino {la solución no está completa}
    hijo.estOpt ← EstimacionOpt(grafo,minArista,hijo.k,nodo.costeT)
    si hijo.estOpt < costeT entonces
        Insertar(hijo, monticulo)
    fsi
fsi
fsi
fpara
fmientras
ffun

```

Una vez creado el nodo inicial, con el que se inicializa el montículo, el algoritmo va tomando nodos, empezando por los de menor coste estimado y explorando sus posibles extensiones. Las extensiones de cada nodo se buscan entre los vértices empezando desde el segundo (el primero se ha seleccionado como nodo de comienzo y final del ciclo), tomando los vértices que no han sido aún asignados a la ruta, y que están conectados con el nodo anterior de la ruta. Para cada uno de ellos se comprueba si se ha alcanzado una solución, es decir se han visitado todos los nodos y el último de la ruta está conectado con el primero. Si se ha alcanzado una solución y ésta es mejor que las anteriores, si las hubiese, se actualizan los datos. El cálculo de la estimación optimista utiliza el valor de la menor arista del grafo, *minArista*, que calcula la función *menorArista* en las inicializaciones.

La función que calcula la estimación optimista para cada nodo parte del valor de los tramos correspondientes a los vértices ya asignados. Para el resto de los tramos de la ruta se toma el valor de la arista de menor coste en el grafo.

```

fun EstimacionOpt (grafo: Tgrafo, minArista: entero, k: entero, costeT: real): real
    estimacion, menorC: real
    estimacion ← costeT + (n-k+1) * minArista
    dev estimacion

```

ffun

Cada nodo del árbol se expande en las $n - k$ asignaciones de vértices pendientes y el árbol de exploración tiene n niveles correspondientes a los n vértices de la ruta. Por lo tanto, el coste del problema está acotado superiormente por el tamaño del árbol, que es del orden de $O(n!)$.

7.4 Selección de tareas: cursos de formación

Un profesional autónomo, que ofrece distintos cursos de formación para empresas y se compromete a haberlos impartido antes de una fecha límite fijada por la empresa, ha recibido n solicitudes de empresas. El autónomo conoce, para cada uno de los cursos solicitados el beneficio b_i que obtendría por impartirlo. El tiempo que se tarda en impartir cada curso es también variable y viene dado por t_i . También sabe los días d_i que quedan antes de la fecha tope fijada por la empresa que ha solicitado cada curso. Se pide un algoritmo que el autónomo pueda utilizar para decidir qué cursos debe escoger para maximizar el beneficio total obtenido.

La fecha límite que fija la empresa para cada curso viene dada por el número de días desde una fecha de referencia, de forma que se trata de una lista d_1, \dots, d_n .

Se trata de un problema de optimización en el que hay dos variables, el beneficio de cada curso y el tiempo requerido para impartirlo. Este es otro de los tipos de problemas que se resuelven con un esquema de ramificación y poda.

Podemos representar la solución mediante un vector *cursos*, donde si la posición i tiene el valor cierto es porque el curso correspondiente a la solicitud i se impartirá. También utilizamos un montículo en el que cada nodo además de almacenar la solución parcial, etapa y cota, almacena el tiempo y beneficio acumulado.

tipo TVectorB = matriz[0..n] de booleano

tipo TVectorR = matriz[0..n] de real

tipo TVector = matriz[0..n] de entero

tipo TNodo = registro

 cursos: TVectorB

 k: entero

 beneficioT: real

 tiempoT: real

 estOpt: real

fregistro

Para que un conjunto de cursos se pueda impartir (sea admisible), la impartición de todos ellos tiene que poder organizarse de forma que cada curso se termine sin superar

el correspondiente límite de días (d_i). Concretamente, si tomamos los cursos i_1, \dots, i_m en este orden, el tiempo que se tarda en terminar de impartir el curso i_k es $\sum_{j=1}^k t_j$, de modo que la secuencia es admisible si $\sum_{j=1}^k t_j \leq d_k$, para todo k entre 1 y m . Para comprobar si un conjunto es admisible ordenamos los cursos por fecha tope creciente y hacemos la comprobación.

Para diseñar el algoritmo de ramificación y poda necesitamos definir las estimaciones pesimista y optimista que utilizará el algoritmo para realizar la poda.

Podemos obtener una estimación optimista sumando el beneficio de todos los cursos que se pueden impartir sin llegar a su fecha límite después de los ya elegidos, como si cada uno se fuera a empezar a impartir justo después de terminar el último impartido.

La estimación pesimista o cota de poda la podemos calcular sumando el beneficio de los cursos ya seleccionados para ser impartidos, y considerando que el resto de los cursos se van impartiendo en orden (por fecha tope creciente) mientras no se supere su fecha límite de impartición. Tras estas consideraciones, el algoritmo puede tomar la siguiente forma:

```

fun Cursos (beneficios: TVectorR, tiempos, limites: TVector,
            cursos: TVertorB, beneficioT: real)

var
    monticulo: TMonticulo
    nodo,hijo: TNodo
    cota,estPes: real

fvar
    monticulo ← CrearMonticuloVacio()
    valor ← 0
    {Construimos el primer nodo}
    nodo.cursos ← cursos
    nodo.k ← 0
    nodo.tiempoT ← 0
    nodo.beneficioT ← 0
    nodo.estOpt ← EstimacionOpt(beneficios, tiempos, limites,
                                  nodo.k,nodo.tiempoT,nodo.beneficioT)
    Insertar(nodo, monticulo)
    cota ← EstimacionPes(beneficios, tiempos, limites,
                          nodo.k,nodo.tiempoT,nodo.beneficioT)
    mientras ¬ MonticuloVacio?(monticulo) ∧
        EstimacionOpt(Primero(monticulo)) > cota hacer
        nodo ← ObtenerCima(monticulo)
        {se generan las extensiones válidas de nodo}
        hijo.k ← nodo.k + 1
        hijo.cursos ← nodo.cursos

```

{ se imparte el curso }

si nodo.tiempoT + tiempos[hijo.k] ≤ limites[hijo.k] **entonces**

 hijo.cursos[hijo.k] ← cierto

 hijo.beneficioT ← nodo.beneficioT + beneficios[hijo.k]

 hijo.tiempoT ← nodo.tiempoT + tiempos[hijo.k]

 hijo.estOpt ← nodo.estOpt

si hijo.k = n **entonces**

si beneficioT < hijo.beneficioT **entonces**

 cursos ← hijo.cursos

 beneficio ← hijo.beneficioT

 cota ← beneficio

fsi

fsi

sino {la solución no está completa}

 Insertar(hijo, monticulo)

fsi

fsi

{ no se imparte el curso }

hijo.estOpt ← EstimacionOpt(beneficios,tiempos,limites,

 hijo.k,nodo.tiempoT,nodo.beneficioT)

si hijo.estOpt ≥ cota **entonces**

 hijo.cursos[hijo.k] ← falso

 hijo.beneficioT ← nodo.beneficioT

 hijo.tiempoT ← nodo.tiempoT

si hijo.k = n **entonces**

si beneficio < hijo.beneficioT **entonces**

 cursos ← hijo.cursos

 beneficio ← hijo.beneficioT

 cota ← beneficio

fsi

fsi

sino {la solución no está completa}

 Insertar(hijo, monticulo)

 estPes ← EstimacionPes(beneficios, tiempos, limites,

 hijo.k,hijo.tiempoT,hijo.beneficioT)

si cota < estPes **entonces**

 cota ← estPes

fsi

fsi

fmiéntras

ffun

Tras inicializar el montículo con el nodo inicial, en el que aún no se ha seleccionado ningún curso para ser impartido, se entra en un bucle que no termina hasta vaciar el montículo o hasta que los nodos que quedan no pueden mejorar la solución actual. En este bucle se van tomando nodos del montículo por orden decreciente de su valor de estimación del beneficio. Para cada nodo se crean sus posibles extensiones, que en este caso consisten en impartir o no impartir el curso correspondiente a la etapa considerada. Para el nodo hijo correspondiente a la opción de impartir el curso no necesitamos recalcular las estimaciones, ya que la impartición del curso ya estaba considerada en la estimación del nodo padre. En este caso se comprueba que se cumpla la restricción de que dicho curso pueda terminarse de impartir antes del plazo límite. Para el nodo hijo correspondiente a la opción de no impartir el curso, sí necesitamos calcular la estimación optimista, para lo que llamamos a la función *estimaciónOpt* utilizando datos del nodo padre en los que no estaba incluido el curso. Para las dos posibles extensiones se comprueba si se ha completado una solución por haber considerado ya todos los cursos, y se actualizan los datos de la mejor solución alcanzada hasta el momento.

Las siguientes funciones se encargan del cálculo de las estimaciones optimista y pesimista respectivamente:

```
fun EstimacionOpt (beneficios, tiempos: TVectorR, limites: TVector,
                    k: entero, tiempoT: real, beneficioT: real): real
var
    estimacion: real
    i: entero
fvar
    estimacion ← beneficioT
para i ← k+1 hasta n hacer
        si tiempoT + tiempos[i] ≤ limites[i] entonces
            estimacion ← estimacion + beneficios[i]
fsi
fpara
dev estimacion
ffun
```

La estimación pesimista también parte del valor del beneficio de los cursos que ya se han seleccionado, y va considerando cursos en orden de tiempos de terminación. Pero en este caso se va sumando el tiempo requerido para impartir los cursos seleccionados al hacer la estimación.

```

fun EstimacionPes (beneficios, tiempos: TVectorR, limites: TVector,
k: entero, tiempoT: real, beneficioT: real): real
  var
    estimacion, tiempo: real
    i: entero
  fvar
    estimacion  $\leftarrow$  beneficioT
    tiempo  $\leftarrow$  tiempoT
    para i  $\leftarrow$  k+1 hasta n hacer
      si tiempo + tiempos[i]  $\leq$  limites[i] entonces
        estimacion  $\leftarrow$  estimacion + beneficios[i]
        tiempo  $\leftarrow$  tiempo + tiempos[i]
      fsi
    fpara
    dev estimacion
ffun

```

Una estimación superior del coste es el tamaño del árbol, que en el peor caso crece como $O(2^n)$, ya que cada nodo del nivel k puede expandirse en las dos opciones de impartir o no el curso, y hay n niveles correspondientes al número de cursos.

7.5 Distancia de edición

Vamos a considerar de nuevo el problema de la distancia de edición que estudiamos en el capítulo dedicado a la programación dinámica. Ahora, a cada operación de transformación le asociamos un coste. Con lo que el planteamiento del problema es el siguiente: Tenemos dos cadenas de caracteres, X e Y , de un alfabeto finito. La cadena $X = x_1, x_2, \dots, x_n$ tiene longitud n , y la cadena $Y = y_1, y_2, \dots, y_m$ tiene longitud m , con $n \leq m$. La cadena X se puede transformar en la cadena Y realizando cambios de los siguientes tipos:

- Borrar un carácter de X , con un coste c_b .
- Insertar uno de los caracteres de Y en X , con un coste c_i .
- Sustituir un carácter de X por uno de los de Y , con un coste c_s .

Nuestro objetivo ahora es encontrar una secuencia de transformaciones que lleven de la cadena X a la cadena Y con un coste asociado mínimo.

Por ejemplo, consideremos las dos siguientes cadenas:

$$\begin{aligned} X &= aabbcb \\ Y &= bbbcb \end{aligned}$$

Una posible transformación sería:

aabbcb → abbcbb borrando la 'a' de la posición 1
 abbcbb → bbbcb cambiando la 'a' de la posición 1 por 'b'

con un coste asociado de $c_b + c_s$.

Consideramos ahora cómo resolverlo aplicando el esquema de ramificación y poda. La idea es ir haciendo transformaciones comenzando por el principio de la cadena X para que sus caracteres coincidan con los de Y. Necesitamos representar las cadenas de caracteres X e Y. Usaremos un montículo de mínimos para almacenar los nodos del espacio de búsqueda e ir tomándolos empezando por los de menor estimación de su coste final. En cada nodo almacenaremos la cadena con las transformaciones realizadas, y la longitud actual, la posición del último carácter que coincide con la cadena destino, una lista de cadenas describiendo las transformaciones realizadas y el coste asociado.

tipo TVectorCad = matriz[0..m+n] de TCadena

tipo TNodo = **registro**

 cadena: TCadena

 long: entero

 k: entero

 transf: TVectorCad

 costeT: entero

 estOpt: entero

fregistro

A continuación consideramos qué estimaciones optimista y pesimista podemos usar para podar el espacio de búsqueda. Una estimación pesimista la podemos obtener sumando al coste de las transformaciones ya realizadas el coste asociado a borrar el resto de los caracteres que quedan de la cadena X más el coste de insertar todos los caracteres que faltan de la cadena Y. Para la estimación optimista podemos suponer que el resto de los caracteres de la cadena X coinciden con los de la cadena Y (sería el mejor caso) y sólo faltaría insertar la diferencia en número de caracteres entre la cadena X y la Y ($m - n$). Para calcular las posibles extensiones válidas de un nodo hay que considerar las posibles situaciones de coincidencia entre la cadena X y la cadena Y:

1. El siguiente carácter coincide en las cadena X e Y, y por tanto no hay que hacer nada.
2. Se borra el siguiente carácter de la cadena X si el siguiente a él coincide con el de la cadena Y. Se actualiza la longitud de la cadena X.
3. Se sustituye el carácter de la cadena X por el de la cadena Y.

4. Se inserta el carácter de la cadena Y, siempre que no se sobrepase la longitud m , y se actualiza la longitud.

En el primer caso no se necesita generar extensiones porque con seguridad ninguna de las otras alternativas va a ser mejor. Pero para el resto de los casos hay que generarlas porque no sabemos cuál puede llevar al coste mínimo.

Con estas ideas formulamos ya un posible algoritmo de ramificación y poda que resuelve el problema, y que utiliza una función, *compleciones*, para crear todas las posibles extensiones del nodo que está considerando. Tras inicializar los datos necesarios el algoritmo comienza creando el nodo inicial, que se mete en el montículo. Después, mientras no se vacía el montículo, y quedan nodos prometedores, se van extrayendo nodos para los que se generan sus extensiones válidas mediante llamadas a la función *compleciones*. Para cada nodo hijo devuelto por esta función se comprueba si se ha alcanzado ya la solución, actualizando los datos si corresponde.

```
fun DistanciaEdición (cadenaX, cadenaY: TCadena, n,m:enteros,
                      transf: TVectorCad, costeT: entero)
```

var

```
  monticulo: TMonticulo
  nodo,hijo: TNodo
  cota,estPes: entero
  listaN: Lista de Tnodo
```

fvar

```
monticulo ← CrearMonticuloVacio()
costeT ← 0
{Construimos el primer nodo}
nodo.cadena ← cadenaX
nodo.long ← 0
nodo.k ← 0
nodo.costeT ← 0
nodo.estOpt ← EstimacionOpt(n,m,nodo.k,nodo.costeT)
Insertar(nodo, monticulo)
cota ← EstimacionPes(n,m,nodo.k,nodo.costeT)
```

mientras \neg MonticuloVacio?(monticulo) \wedge EstimacionOpt(Primero(monticulo)) < cota **hacer**

```
nodo ← ObtenerCima(monticulo)
```

{se generan las extensiones válidas de nodo}

```
listaN ← Compleciones(cadenaX, cadenaY,n,m,nodo)
```

mientras \neg ListaVacio(ListaN) **hacer**

```
  hijo ← Primero(ListaN)
```

si hijo.k = m **entonces**

si costeT > hijo.costeT **entonces**

```

transf ← hijo.transf
costeT ← hijo.costeT
cota ← costeT
fsi
sino {la solución no está completa}
    hijo.estOpt ← EstimacionOpt(hijo.long,m,hijo.k,nodo.costeT)
    Insertar(hijo, monticulo)
    estPes ← EstimacionPes(hijo.long,m,hijo.k,hijo.costeT)
    si cota > estPes entonces
        cota ← estPes
    fsi
    fsi
fmientras
fmientras
ffun

```

La función *compleciones*, que almacena los hijos resultantes de las posibles extensiones en una lista, comprueba en primer lugar si en la posición que se está considerando las dos cadenas coinciden. Si es así sólo se genera una extensión que corresponde a desplazar la posición en curso al siguiente carácter. Si los caracteres no coinciden, se generan las extensiones correspondientes a las posibles operaciones: borrado (si el siguiente carácter al borrado coincide con el buscado), sustitución, e inserción (si no se sobrepasa la longitud de la cadena destino).

```

fun Compleciones (cadenaX, cadenaY: TCadena, n,m:enteros,
                    nodo: TNodo): Lista de TNodo
var
    hijo: TNodo
    listaN: Lista de TNodo
fvar
    listaN ← CrearLista()
    hijo.k ← nodo.k + 1
    hijo.cadena ← nodo.cadena
    hijo.transf ← nodo.transf
    si cadenaX[hijo.k] = cadenaY[hijo.k] entonces
        hijo.costeT ← nodo.costeT
        hijo.estOpt ← nodo.estOpt
        hijo.long ← nodo.long
        listaN ← Añadir(listaN, hijo)
    sino
        {borrado}
        si cadenaX[hijo.k+1] = cadenaY[hijo.k] entonces

```

```

para i ← hijo.k hasta hijo.long-1 hacer
    hijo.cadena[i] ← nodo.cadena[i+1]
fpara
    hijo.costeT ← nodo.costeT + costeBorrado
    hijo.transf[hijo.k] ← “borrado posición” + hijo.k
    hijo.long ← nodo.long - 1
    listaN ← Añadir(listaN, hijo)
fsi
{ sustitución}
hijo.cadena[hijo.k] ← nodo.cadena[hijo.k]
hijo.costeT ← nodo.costeT + costeSustitución
hijo.transf[hijo.k] ← “sustituido posic” + hijo.k+ “carácter”+nodo.cadena[hijo.k]
hijo.long ← nodo.long
listaN ← Añadir(listaN, hijo)
{ inserción}
si nodo.long < m entonces
    para i ← hijo.k+1 hasta hijo.long+1 hacer
        hijo.cadena[i] ← nodo.cadena[i-1]
    fpara
        hijo.cadena[hijo.k] ← nodo.cadena[hijo.k]
        hijo.costeT ← nodo.costeT + costeInserción
        hijo.transf[hijo.k] ← “inserción posic” + hijo.k+ “carácter”+nodo.cadena[hijo.k]
        hijo.long ← nodo.long + 1
        listaN ← Añadir(listaN, hijo)
fsi
fsi
dev listaN
ffun

```

Las funciones que calculan las estimaciones optimista y pesimista pueden adoptar la siguiente forma:

```

fun EstimacionOpt (lonX, lonY,k,costeT:enteros): entero
    var
        estimacion: entero
    fvar
        estimacion ← costeT + costeInserción * (lonY-lonX)
        dev estimacion
ffun

```

Como explicábamos antes, la estimación optimista supone que todos los caracteres que faltan por comprobar entre las dos cadenas coinciden, y que sólo es necesario insertar los que faltan para llegar a la longitud de la cadena destino. La estimación pesimista supone que es necesario borrar todos los que quedan por comprobar e insertar los de la otra cadena:

```
fun EstimacionPes (lonX, lonY,k,costeT:enteros): entero
```

```
var
```

```
estimacion: entero
```

```
fvar
```

```
estimacion ← costeT + costeBorrado * (lonX-k)
```

```
estimacion ← estimacion + costeInserción * (lonY-k)
```

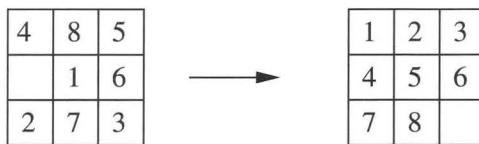
```
dev estimacion
```

```
ffun
```

Buscamos una cota superior del coste del algoritmo analizando el espacio de búsqueda. En el caso peor se generan 3 hijos por nodo hasta llegar a una profundidad igual a la longitud de la cadena destino m . Por lo tanto, una cota superior al número de nodos a explorar es 3^m .

7.6 Ejercicios propuestos

- Se pide un algoritmo para resolver el puzzle que se describe a continuación. Se dispone de un tablero de n^2 casillas en las que hay $n^2 - 1$ piezas que están numeradas del 1 al $n^2 - 1$, de forma que a una de las casillas le corresponde un *hueco*. Para resolver el puzzle hay que mover las piezas para transformar la configuración inicial en una disposición en la que las piezas están ordenadas por filas, es decir, a la casilla (i, j) le corresponde la pieza $(i - 1)*n + j$, y el hueco se encuentra en la última casilla (n, n) , como muestra la figura:

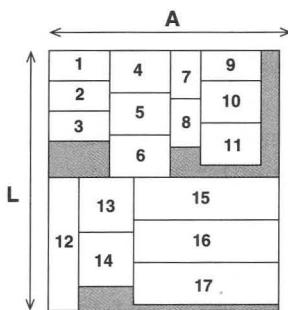


Los movimientos válidos son los de las piezas adyacentes al hueco que pueden ocuparlo (arriba, abajo, izquierda y derecha), pasando a estar el hueco en el lugar de la pieza movida.

- Se quiere realizar en un soporte secuencial (cinta de T minutos en cada cara) una recopilación de canciones preferidas. Se dispone de una lista de n canciones favoritas, junto con la duración individual de cada una. Lamentablemente, la cinta

no tiene capacidad para contener todas las canciones, por lo que se les ha asignado una puntuación (cuanto mayor es la preferencia, mayor es la puntuación). Se pide un algoritmo que obtenga la mejor cinta posible según la puntuación, teniendo en cuenta que las canciones deben caber enteras y no es admisible que una canción se corte al final de una de las caras.

- En un problema de cortado de patrones se necesitan n piezas planas rectangulares p_1, \dots, p_n , donde las dimensiones (ancho, alto) de la pieza i son (x_i, y_i) . Estas piezas han de extraerse de un tablero de material de forma rectangular, como en el ejemplo que muestra la figura:



Se pide un algoritmo que encuentre una colocación de las piezas de forma que se minimice el área del tablero de material que se necesita.

- Se tienen n huertos, cada uno con un tipo de cultivo diferente. Los cultivos ya están maduros y es necesario recolectar los frutos. Para cada huerto i , siendo $1 \leq i \leq n$, se conoce el valor v_i del fruto que se puede recolectar en él. Se conoce asimismo, el tiempo t_i que se tarda en recolectar el fruto del huerto i , y el número de días d_i que tardan en echarse a perder los frutos de cada huerto. Se pide un algoritmo que seleccione los huertos en los que hacer la recolección antes de que se echen a perder los frutos, de forma que se maximice el valor de los frutos recolectados.
- Una empresa de montajes tiene n montadores con distintos rendimientos según el tipo de trabajo. Se pide un algoritmo que asigne los próximos n encargos, uno a cada montador, minimizando el coste total de todos los montajes. Se conoce de antemano la tabla de costes $C[1..n, 1..n]$ en la que el valor c_{ij} corresponde al coste de que el montador i realice el montaje j .
- Se dispone de un conjunto de sellos de n valores diferentes, habiendo en el conjunto 3 sellos de cada tipo. En el sistema de franqueo del país sólo se admite un máximo de 5 sellos por carta. Se pide un algoritmo que seleccione de este conjunto los sellos para enviar una carta cuya tarifa de envío es E , de forma que el gasto en sellos sea mínimo (con un franqueo igual o superior a la tarifa E).

7. Consideremos una variante del problema de la mochila, en la que en lugar de tener n objetos distintos, se dispone de n tipos de objetos, pudiéndose escoger tantas unidades como se desee de cualquiera de los tipos. Se pide un algoritmo que realice una selección de objetos que maximice el valor del contenido de la mochila.
8. Se dispone de un conjunto de n objetos blandos, que adquieren la forma del espacio que los contiene, pero que no se pueden fraccionar. Cada objeto o_i ($1 \leq i \leq n$) de este conjunto tiene un volumen v_i . Se dispone también de una cantidad ilimitada de envases de volumen C . Se pide un algoritmo que calcule la forma óptima de empaquetar los n objetos en recipientes de forma que se minimice el número de envases necesarios.
9. En cierto país se utiliza un sistema monetario formado por n tipos de monedas distintos, de valores v_1, \dots, v_n . Sabiendo que de cada tipo de moneda i se dispone de una cantidad de monedas c_i , se pide un algoritmo que calcule las monedas de cada tipo que hay que utilizar para pagar una cantidad C minimizando el número total de monedas.
10. Se dispone de un conjunto de n componentes electrónicas que deben colocarse sobre n posiciones de una placa. Se tienen como datos dos matrices N y D , de dimensiones $n \times n$. El elemento (i, j) de la matriz N indica el número de conexiones necesarias entre la componente i y la componente j . El elemento (i, j) de la matriz D indica la distancia en la placa entre la posición i y la posición j . Un cableado de la placa, c_1, \dots, c_n , es una asignación de componentes a posiciones de la placa. Si la longitud del cableado se establece por la siguiente fórmula:

$$\sum_{i < j} N[i, j]D[c_i, c_j]$$

se pide un algoritmo que busque el cableado de longitud mínima.

7.7 Notas bibliográficas

Land y Doig [LD60] propusieron en 1960 la técnica de ramificación y poda en el contexto de la programación lineal. Lawler y Wood [LW66] publicaron uno de los primeros trabajos detallados dedicados al tema y Bellmore y Nemhauser [Bel68] resolvieron el problema del viajante de comercio aplicando esta técnica.

Ejemplos de aplicación del esquema de ramificación y poda pueden encontrarse en diversos libros didácticos de algoritmia [Gue00, MOOMV03, GMCLMPGC03]. En [Gue00] se pueden encontrar entre otros, el problema del puzzle, el cortado de patrones y la mochila con múltiples elementos. En [MOOMV03] se resuelven entre otros los problemas de la colección de canciones, la recolección de huertos, el franqueo de cartas y el

empaque de objetos. En [GMCLMPGC03] se resuelven entre otros el problema del cortado de patrones. Ejemplos de problemas de asignación de tareas pueden encontrarse en todos los libros citados.

Bibliografía

- [AHU98] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Estructuras de datos y algoritmos*. Addison Wesley Iberoamericana, 1998.
- [AP89] J. Amstel and J. Poirters. *The design of data structures and algorithms*. Prentice Hall, 1989.
- [BB90] B. Brassard and P. Bratley. *Algorítmica: Concepción y Análisis*. Masson, S.A., 1990.
- [BB06] G. Brassard and P. Bratley. *Fundamentos de algoritmia*. Prentice-Hall, 2006.
- [Bel57] Richard Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, 1957.
- [Bel68] Bellmore, M. and Nemhauser, G. L. The traveling salesman problem : A survey. *Operations Res.*, 16:538–558, 1968.
- [BM67] E. O. Brigham and R. E. Morrow. The fast fourier transform. *Spectrum, IEEE*, 4(12):63 –70, dec 1967.
- [Chr75] Nicos Christofides. *Graph Theory. An Algorithmic Approach*. Academic Press, 1975.
- [CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.
- [Cob65] A. Cobham. The intrinsic computational difficulty of functions. In Y. Bar-Hillel, editor, *Proceedings of the 1964 International Conference for Logic, Methodology, and Philosophy of Science*, pages 24–30. North-Holland, 1965.
- [DDH72] O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. *Structured Programming*. Academic Press, London, 3 edition, 1972.

- [Dij59] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.
- [Dij68] E. Dijkstra. Go-to statement considered harmful. *Comm. ACM*, 11(5), 1968.
- [Dij72] E. W. Dijkstra. Notes on structured programming. In *Structured Programming*. Academic Press, 1972.
- [Edm71] Jack Edmonds. Matroids and the greedy algorithm. *Mathematical Programming*, 1:126–136, 1971.
- [Flo62] Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5:345–, June 1962.
- [Flo67] Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Symposia in Applied Mathematics*, pages 19–32. American Mathematical Society, 1967.
- [FT87] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34:596–615, July 1987.
- [GA97] Miguel Gonzalo Arroyo, Julio y Rodríguez Artacho. *Esquemas algorítmicos. Enfoque metodológico y problemas resueltos*. UNED, 1997.
- [GA01] Julio Gonzalo Arroyo. *Proyecto Docente*. UNED, 2001.
- [GB65] S. W. Golomb and L. D. Baumert. Backtrack programming. *Jrnl. A.C.M.*, 12(4):516–524, October 1965.
- [GMCLMPGC03] Ginés García Mateos, Joaquín Cervera López, Norberto Marín Pérez, and Domingo Giménez Cánovas. *Algoritmos y Estructuras de Datos*. ICE, Universidad de Murcia, 2003.
- [GP70] Richard Gauthier and Stephen Ponto. *Designing Systems Programs*. Prentice Hall, Englewood Cliffs, 2 edition, 1970.
- [Gri82] David Gries. A note on a standard strategy for developing loop invariants and loops. *Science of Computer Programming*, 2(3):207–214, 1982.
- [Gue00] Antonio Guerequeta, Rosa y Vallecillo. *Técnicas de diseño de algoritmos*. Universidad de Málaga, 2000.

- [Gut75] J. Guttag. *The specification and application to programming of abstract data types*. PhD thesis, Univ. of Toronto, Dept. of Computer Sciences, 1975. TR CSRG-59.
- [Hoa62] C. Hoare. Quicksort. *Computer Journal*, 5(1), 1962.
- [Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.
- [HS94] E. Horowitz and S. Sahni. *Fundamentals of data Structures in Pascal*. Computer Science Press, 1994.
- [Knu73] D. Knuth. *The Art of Computer Programming vol. III. Sorting and Searching*. Addison-Wesley, 1973.
- [Knu76] D. E. Knuth. Big omicron and big omega and big theta. *SIGACT Automata and Computability Theory*, 1976.
- [Krz81] R. Apt Krzysztof. Ten years of hoare’s logic. *ACM Transactions on Programming Languages and Systems*, 3:431–483, 1981.
- [Krz86] R. Apt Krzysztof. Correctness proofs of distributed termination algorithms. *ACM Transactions on Programming Languages and Systems*, 8(3):388–405, July 1986.
- [LD60] A. H. Land and A.G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520, 1960.
- [LW66] E. L. Lawler and D. E. Wood. Branch-and-bound methods: A survey. *Operations Research*, 14(4):699–719, 1966.
- [McC63] J. McCarthy. Towards a mathematical theory of computation. In *Proc. IFIP Congress 62*, pages 21–28, Amsterdam, 1963. North-Holland.
- [MOOMV03] Narciso Martí Oliet, Yolanda Ortega Mallén, and Alberto Verdejo. *Estructuras de datos y métodos algorítmicos: Ejercicios resueltos*. Colección Prentice Practica. Pearson/Prentice Hall, 2003.
- [NS63] A. Newell and H. A. Simon. GPS: A program that simulates human thought. In E. A. Feigenbaum and J. Feldman, editors, *Computers and Thought*. McGraw-Hill, New York, 1963.
- [OG76] Susan Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM*, 19(5):279–285, 1976.

- [Owi76] Susan Owicky. A consistent and complete deductive system for the verification of parallel programs. In *Conference Record of the Eighth Annual ACM Symposium on Theory of Computing*, pages 73–86, Hershey, Pennsylvania, 3–5 May 1976.
- [Par72] David Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.
- [ST81] M.N.S. Swamy and K. Thulasiraman. *Graphs, Networks, and Algorithms*. John Wiley & Sons, 1981.
- [Tur49] Alan M. Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69, Cambridge, England, June 1949. University Mathematical Laboratory.
- [Wil64] J. Williams. Algorithm 232: Heapsort. *Comm. ACM*, 1964.
- [Wir71] N. Wirth. Program development by stepwise refinement. *Comm. ACM*, 14(4), 1971.
- [Wir82] N. Wirth. *Programming in Modula-2*. Springer-Verlag, 1982.

Apéndice A

Notación

- Funciones:

fun (parámetros): valores devueltos
instrucciones
dev valor devuelto

ffun

No se distingue entre parámetros de entrada y salida, que se identifican por el contexto.

- Asignación: \leftarrow

- Instrucciones condicionales:

– **si** condición **entonces**
instrucción
fsi

– **si** condición **entonces**
instrucción1
sino
instrucción2
fsi

– **caso de**
condición1 **hacer**
instrucción1
condición2 **hacer**
instrucción2
...

condiciónN hacer
instrucciónN
fcaso

- Instrucciones iterativas:

- **mientras** condición **hacer**
 instrucciones
fmientras
- **para** contador \leftarrow valor1 **hasta** valor2 **hacer**
 instrucciones
fpara
- **para** contador \leftarrow valor1 **hasta** valor2 **incremento - 1 hacer**
 instrucciones
fpara

- Tipos de datos:

tipo Nombre = declaración

Utilizamos mayúsculas para los nombres de tipos.

- Vectores:

matriz[rango] **de** tipo básico

- Matrices:

matriz[rango1, rango2, ..., rangoN] **de** tipo básico

- Registros:

registro

campo1: tipo1

campo2: tipo2

...

fregistro

- Variables:

var

nombreVariable: tipoVariable

...

fvar

- Operadores Lógicos:

- y: \wedge

- o: \vee

- no: \neg

- Operadores matemáticos:

- +, -, *, /

- División entera: **div**

- Función módulo: **mod**

- Función valor absoluto: **abs**