Autor, Carles Franquesa i Niubò, *carlesfranquesa@gmail.com*
Diseño de la portada, www.albertclaret.com, *info@albertclaret.com*
Ilustraciones, Santamaria, *enriccastellvi@yahoo.es*
Producción, CPET S.C.C.L.

Depósito legal B-42100-2010
ISBN 978-84-938406-0-0
Primera Edición en castellano. Impreso en Barcelona, diciembre 2010.

El código fuente que se muestra bajo el título genérico de *Algoritmo* a lo largo de todo el libro, se puede solicitar por correo electrónico al mismo autor. La relación de todos los algoritmos se encuentra en las últimas páginas, entre las relaciones de figuras y de esquemas algorítmicos.

algoritmia comentada

$$cómo = \int_{?}^{!} qué(t)\ dt$$

carles franquesa i niubò

qué es a cómo lo que cuántos es a cuáles.

Muhammad ibn Musa al-Jwarizmi (al-Khwa-rizmi)

Prólogo

Traducción del prólogo a la versión original

La Informática es una amalgama compleja de ciencia y tecnología. Dentro de su vertiente científica, la Algoritmia, la ciencia que estudia los algoritmos y sus propiedades de corrección y eficiencia, ocupa un lugar de excepción, a pesar de que, como la mayor parte de los fundamentos científicos de la informática, es una gran desconocida del público.

La Algoritmia se nutre de las matemáticas, especialmente de la combinatoria y la matemática discreta, y la lógica, y en su seno se han desarrollado una gran variedad de técnicas, conceptos y resultados innovadores y fascinantes. En muchos sentidos la Algoritmia es el corazón de la Informática, y también ha cambiado profundamente muchas áreas de las matemáticas. Ha introducido como un objetivo importante de toda investigación matemática el desarrollo de soluciones efectivas y eficientes, es decir, que los problemas se puedan resolver mecánicamente, con un ordenador, y de manera rápida.

Del papel fundamental de la Algoritmia en la Informática es testimonio la enorme cantidad de libros de texto que cada año aparecen sobre algoritmos y estructuras de datos, y su inclusión en todos los currículums universitarios de Informática del mundo desde los años 70. Con este panorama, un libro como el que tenéis en las manos podría pasar desapercibido, uno más de tantos. Sin embargo, hay dos elementos muy significativos que no encontraréis en los otros libros de texto sobre la materia.

Por una parte, es uno de los pocos libros que se han escrito en nuestro lenguaje[1]*. Como fácilmente podréis imaginar la inmensa mayoría de los libros que se han escrito sobre Algoritmia, lo han sido en inglés. Ocasionalmente, se han hecho traducciones del inglés a otras lenguas mayoritarias como el español, el alemán o el francés. Pero no conozco ninguna traducción al catalán. Así pues, la publicación de un libro de texto universitario sobre Algoritmia en catalán es una noticia excelente por la cual nos hemos de congratular.*

[1] NT: Se refiere al catalán.

El otro elemento significativo que aporta "Algoritmia Comentada" es la enorme capacidad pedagógica y el entusiasmo desbordante de Carles Franquesa. Hace apenas diez años que nos conocimos, y durante muchos de ellos coincidimos como profesores de la asignatura "Análisis y Diseño de Algoritmos" en la Facultad de Informática de Barcelona. La pasión por la Algoritmia ha sido siempre un nexo de unión entre nosotros, y hemos mantenido muchísimas discusiones enriquecedoras a lo largo de todos estos años. Tanto el uno como el otro hemos vivido y vivimos una relación constante con esta ciencia fascinante, como docentes y también como investigadores, pues ambos trabajamos en temas de Algoritmia.

Página a página, el amor y el entusiasmo de Carles por la Algoritmia se hacen visibles. Como libro de texto escrito por una persona que conoce la materia, encontraréis todo lo que uno puede esperar. Como libro de texto escrito por una persona que ama profundamente la materia, encontraréis algo nuevo. Más que una fría y rigurosa exposición, encontraréis una conversación, un diálogo, seguiréis el tren de pensamientos del autor, redescubriréis de su mano los conceptos, las vías exitosas y los caminos sin salida, aprenderéis a pensar "algoritmicamente".

Pero, por encima de todo, confío en que Carles haya sido capaz de transmitiros y contagiaros este entusiasmo por la Algoritmia.

Conrado Martínez
Barcelona, 25 de enero de 2010

Prólogo a la versión en español

Hace cosa de ocho meses salía a la luz el libro Algorísmia Comentada *de Carles Franquesa, quien tuvo la amabilidad de pedirme que lo prologara. Ahora, para esta nueva edición en español, Carles me ha vuelto a pedir que escribiera una presentación. Destacaba en mi prólogo de la edición original en catalán dos aspectos: Por un lado, la buena noticia que suponía la aparición de un libro de texto universitario sobre esta materia escrito en catalán; por otro lado, el entusiasmo y el enfoque refrescante que Carles transmitía en cada una de las páginas del libro. Y ambos aspectos son también destacables en esta presente edición, por razones que me apresuro a exponer.*

Si bien la oferta de libros de texto universitarios sobre Algorítmica es más extensa en castellano, no es menos cierto que en la mayoría de los casos son traducciones de libros originalmente escritos en inglés. Además es usual que las traducciones dejen bastante que desear, bien porque el traductor no es un experto en la materia sobre la que versa el libro, bien porque la traducción la lleva a cabo alguien versado en la materia, pero sin una preparación profesional como traductor. Y en definitiva, no abundan los libros sobre Algorítmica es-

critos por autores españoles o hispanoamericanos. De manera que nuevamente es un motivo de satisfacción la aparición de este libro de Carles Franquesa en castellano. Y aunque, en gran parte, se trata de una traducción del original en catalán, ésta ha sido realizada por el propio autor, lo cual es una garantía de fidelidad y de calidad.

Igual que en su predecesora, el enfoque eminentemente pedagógico, el entusiamo de Carles al enseñarnos una materia que ama profundamente se destila en cada página de esta nueva edición. La lengua, sea catalán o español, no es una barrera sinó todo lo contrario, un vehículo para la transmisión del saber. Tal como indicaba en mi primer prólogo, como libro escrito por un buen conocedor de la materia, en Algoritmia Comentada *encontraréis todo lo que cabría esperar. Sin embargo, también conoceréis mucho más. Lejos de una fría y rigurosa exposición de conceptos, os sumergiréis en un diálogo con el autor, seguiréis sus —a veces intrincadas, como la vida misma— líneas de razonamiento. Redescubriréis los conceptos más fundamentales de la disciplina acompañados de su mano, aprenderéis a pensar "algorítmicamente". En una exploración constante, sabréis de las vías que conducen al éxito en la resolución de un problema y también a los callejones sin salida. Para cuando hayáis leído el libro sabréis que esto se llama en el argot* algoritmos de vuelta atrás*, aunque frecuentemente se usa el término original en inglés* algoritmos de backtracking*.*

A lo largo de este prólogo me he referido al libro que tienes entre manos como una nueva edición. El término está usado con propiedad, pues no se trata de una mera traducción del original en catalán. Carles ha aprovechado la ocasión para introducir algunas mejoras y desarrollar en mayor profundidad algunos temas, en particular, el de Ramificación y Poda *(Sección 7.3), un tema especialmente querido y próximo para Carles, pues su trabajo de investigación se centra en los problemas de optimización combinatoria y su resolución mediante la programación lineal y la ramificación y poda, entre otras técnicas. Así que, agotada la primera tirada de la edición en catalán, es muy posible que Carles se anime a preparar una segunda edición de* Algorísmia Comentada*, esta vez traduciendo las novedades de la edición en castellano al catalán, y cerrando así el círculo.*

Para concluir, confío en que Carles sea capaz nuevamente de transmitiros y contagiaros su entusiasmo ilimitado por esta materia fascinante: la Algorítmica.

Conrado Martínez
Barcelona, 10 de Septiembre de 2010

Índice

Preámbulo

Buscando el origen de la palabra, la etimología, en el Diccionario de la Real Academia Española [12], nos encontramos con la definición

algoritmia *(De algoritmo.) f. Ciencia del cálculo aritmético y algebraico; teoría de los números.*

Por curiosidad, mirando en la wikipedia descubrimos un matemático, físico, astrónomo y cartógrafo persa de hace ya mil dos cientos años, conocido por el nombre de Al-Khwa-rizmi. Se pronunciaría *alguarismi*, más o menos. De hecho, el término *guarismo*, que significa "cifra", procede de la misma raíz, tal como dicta también el mismo diccionario. Así pues, todo apunta a que este señor tan sabio escribió el primer texto de álgebra. Parece ser que la única imagen que de él tenemos es este sello ruso que hay justo antes del prólogo del Profesor Conrado Martínez. El sello es de 1983 conmemorando su 1200 aniversario, tal como dice impreso.

Bien, volviendo a las definiciones, parece que no íbamos desencaminados. Ubicar el algoritmia dentro del álgebra resulta razonable. Aún así, la definición del diccionario oficial resulta demasiado genérica para el enfoque que aquí se pretende dar. Por eso, mejor orientamos el discurso tomando por referencia otro concepto de la misma raíz.

algoritmo *1. m. Conjunto ordenado y finito de operaciones que permite hallar la solución de un problema. 2. m. Método y notación en las distintas formas del cálculo.*

Definitivamente, partiremos de la primera de estas dos acepciones. Entonces, definimos aquí el algoritmia como la ciencia que estudia los algoritmos. Por ciencia entendemos toda disciplina que utiliza los modelos matemáticos para conseguir sus propósitos.

Vamos a echar un vistazo a un tema profundamente interesante. Dejando aparte la inmensa mayoría de los problemas que tenemos, hay algunos que pretendemos estudiar en este libro. Los llamamos problemas *computacionales*, caracterizando con este adjetivo todos aquellos que pueden ser resueltos mediante

una máquina computadora. Esta es una referencia clave que convendría no perder de vista a lo largo de este viaje. Vamos a estudiar problemas. Problemas computacionales.

En la definición se habla de un problema, sin mencionar referencia alguna a los datos que lo materializan. O sea, que vamos a entender que sumar uno más uno es el mismo problema que sumar uno más otro. Aquí también consideraremos un problema como algo independiente de los datos que lo formalizan. Nos disponemos a trabajar con los problemas sin tener en cuenta el valor de los datos concretos que requieren para ser completamente enunciados.

En cierta manera es un poco contradictorio. Parece que no se puede ignorar una parte tan esencial de un concepto cuando nos disponemos a estudiar su todo.

Sin embargo, por otra parte, resulta comprensible hablar de problemas sin plantear ningún ejemplo concreto. Y es por ese camino hacia donde parece conveniente arrancar nuestro estudio, ya que si pretendiéramos tener en cuenta el conjunto de todos los datos posibles, para cada problema, nunca podríamos dar el primer paso para estudiar su complejidad.

Como siempre se hace en los estudios analíticos, cuando un factor decisivo tiene un amplio margen de variabilidad y se pretende ignorar, hay que recurrir a la estadística. Para seguir este libro no hace falta gran conocimiento estadístico. Hay alguna referencia a distribuciones de probabilidad uniforme, pero nada más. Lo que más se utilitzará son los conceptos de *caso peor*, *caso medio* y, no tan frecuentemente, *caso mejor*.

En definitiva estudiaremos problemas. Ahora bien, para disponer de unos contenidos más o menos reglados (de regla), es necesario comprender la estrecha relación que hay entre estudiar y tomar medidas.

La finalidad del algoritmia consiste en medir la dificultad que representa solucionar problemas.

Normalmente utilizaremos el vocablo *complejidad* para referirnos a dificultad. Es cuestión de convenio. Históricamente ha habido el consenso de hablar de complejidad que ciertamente parece un término más riguroso que dificultad, que tiene connotaciones más subjectivas. Y está claro que si nuestra intención es tomar medidas de alguna cosa, en ningún caso nos convendrá introducir ningún tipo de subjectividad.

También nos podemos referir al mismo concepto de complejidad con otra palabra que tiene una connotación diametralmente antagónica, *eficiencia*.

A ver si lo adiviná is... ¿Cuándo nos referiremos de una forma y cuándo de la otra?. La respuesta está perdida por el libro...

La fórmula de la portada de este libro,

$$\textit{cómo} = \int_{?}^{!} \textit{qué(t)}\ dt$$

se refiere a cómo se hacen las cosas, y qué se hace en cada instante. Se pronuncia diciendo *cómo es igual a la integral, desde el instante interrogante hasta la admiración, de qué de té diferencial de té*, y significa que la manera de hacer una cosa, el cómo, es una suma de infinitas cosas que hay que hacer desde el momento en que comenzamos hasta que acabamos.

Por ejemplo, cómo se hace una tortilla de patatas comienza cogiendo las patatas. Coger las patatas es lo qué se hace. Y después, en el siguiente diferencial de tiempo, lo qué se hace es pelarlas. Y después... En fin, que qué se hace en cada momento explica cómo se hace la cosa. El tiempo viene acotado desde el principio, indicado por el interrogante de conseguir el objectivo, hasta la admiración, cuando se ha cumplido.

Es curioso pensar que después de todo, en alguna dimensión por encima, hacer la tortilla de patatas también puede ser considerado como un solo qué. ¿Qué has hecho para comer? Tortilla de patatas. Y también hacia la otra dirección, está bien claro. En ese caso vamos a parar a alguna dimensión inferior. Un qué de los de antes, coger las patatas, se puede transformar en un cómo cogerlas. Y entonces, este cómo tiene otros qués más pequeñitos, por ejemplo, abrir la bolsa donde están.

En resumidas cuentas, está bien claro que la analogía entre el lenguage verbal y los lenguages informáticos es estrecha. Qué es una llamada a una rutina. Cómo es la implementación del cuerpo de la rutina.

Este libro sigue el temario que he impartido en la materia de Análisis y Diseño de Algoritmos, en la Facultad de Informática de Barcelona, de la Universitat Politècnica de Catalunya, durante varios años. Así pues, se trata en última instancia de un libro académico, y se supone al lector ciertos conocimientos de programación de computadoras.

Va dirigido a estudiantes universitarios de carreras técnicas, estudiantes de grado en informática, matemáticas, e ingenierías en general.

Los fragmentos de código que hay a lo largo de todo el texto bajo el título genérico de *Algoritmo*, se encuentran también en forma de archivos de código fuente. El autor se compromete a enviar un archivo comprimido con este contenido a todo aquél que lo solicite, por correo electrónico a la dirección que se indica en las referencias de este libro. Hay un programa principal para cada capítulo del libro. El lenguage de programación utilizado es C++, con la librería *std*. No obstante, de esta librería sólo se utiliza la serialización con *cin »* y *cout «* en los programas principales. Es decir, los tipos de datos que se manejan son tipos primitivos del lenguaje C, aunque hay implementaciones que utilizan clases. Por esta razón, se recuerda que una clase es una estructura que, además de variables y funciones miembro, define un espacio de visibilidad. En general se utiliza el concepto de estructura para las clases pequeñas. La diferencia entre estructura y clase es que las estructuras lo tienen todo público, en cambio las clases pueden tener variables y funciones privadas. En definitiva, se espera del lector que conozca algunas instrucciones de gestión de memoria (*new*, *delete*, *memset* o *memcpy*), la función *sizeof*, algunas funciones básicas de vectores de caracteres, como *strlen*, *strcpy* y poca cosa más. En los últimos capítulos se utilizan plantillas para definir las estructuras dinámicas.

Capítulo 1

Notación Asintótica

Supongamos que los problemas tienen una complejidad consustancial. En otras palabras, establecemos la hipótesis de que la dificultad que tenemos los seres humanos para resolver un problema depende exclusivamente del problema en sí, y no de nuestra capacidad para resolverlo. Suponemos, en definitiva, que todos somos igual de sabios. Actuamos así a pesar de que no lo sepamos demostrar. Sencillamente porque es un modelo que nos vale. Probablemente nos estemos equivocando, y cualquier problema, por difícil que nos parezca, pueda ser resuelto con métodos rápidos y sencillos. Pero mientras no nazca alguien que nos explique la manera de hacerlo, esta suposición nos resulta lo bastante útil.

Para estudiar los problemas nos hace falta alguna herramienta que nos permita medir su complejidad. No estamos hablando de cosas raras. De la misma manera que un carpintero necesita un metro para poder medir la longitud o la anchura de un tablón, en el ámbito del algoritmia necesitamos alguna referencia común para todos los problemas. Un mecanismo para poder equiparar complejidades. Entonces podremos decir que un problema es igual de difícil de resolver que otro. O más. O menos.

En este capítulo se presentan primero los conceptos matemáticos indispensables para comprender el significado de la notación asintótica. A partir de la tercera sección, el texto se orienta a mostrar la necesidad de esta herramienta para nuestros propósitos. Entonces se define la notación asintótica, O, Θ y Ω (llamadas "o", "zeta" y "omega"), para acabar con algunas secciones donde se tratará de aprender a utilizarla como instrumento de medida para las complejidades de los problemas computacionales.

1.1 Preliminares

1.1.1 Inducción

La inducción matemática es una componente esencial en la formalización de la teoría de conjuntos. Los axiomas de Peano establecen la definición de los números naturales. Son cinco postulados. En el quinto, se define la inducción matemática. Resulta sorprendente la analogía entra ésta, y la teoría cristiana.

Axiomas de Peano

Giuseppe Peano

Giuseppe Peano (1858-1932) fue un matemático nacido en Piamonte que estableció los cinco axiomas que definen formalmente el conjunto de los números naturales. Además, también fue el introductor de una parte importante de la notación matemática actual, así como el primero en utilizar los símbolos de la unión ($\cup$) o la intersección ($\cap$) de conjuntos.

Jugar a satisfacer los axiomas de Peano resulta bastante ilustrativo para comprender que, en un sistema de símbolos, no hay otra manera de satisfacer los cuatro primeros axiomas que no sea definiendo los números naturales. Con un papel y un lápiz seguimos las reglas de los axiomas como se explica a continuación.

Los cuatro primeros axiomas son los siguientes:

Axioma primero. El número 1 existe (cosa que recuerda a amar a dios sobre todas las cosas). Es decir, el conjunto de los números naturales no es vacío. En este momento dibujamos en el papel un número 1 en un círculo, Figura 1.1.

Figura 1.1: *Axioma I. El número 1 existe.*

Axioma segundo. Cualquier número tiene asociado un sucesor que también es un número (cosa que recuerda a amar al prójimo como a uno mismo). Hacemos salir una flecha del círculo donde hay el 1 para representar al sucesor, y dibujamos otro círculo con un símbolo cualquiera (por ejemplo un 2) en su interior.

Ojo! Así no estamos respetando este segundo axioma!. El nuevo círculo (que también es un número según este segundo axioma) necesita un sucesor. Para satisfacerlo con el mínimo esfuerzo, hacemos que el sucesor del número 2 sea el 1, Figura 1.2. Así, el sucesor del 1 es el 2, y el del 2 es el 1. Todo va bien.

Figura 1.2: *Axioma II. Cualquier número tiene asociado un sucesor que también es un número.*

Axioma tercero. Ningún número tiene por sucesor al 1. O sea, que mal. Así pues, tachamos la segunda flecha dibujada antes, y nos inventamos un nuevo símbolo, por ejemplo el 3. Entonces, podemos poner una flecha para indicar que el sucesor del 2 es el 3, y el sucesor del 3, el 2, Figura 1.3. Seguimos satisfaciendo los axiomas vistos hasta ahora.

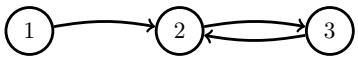


Figura 1.3: *Axioma III. Ningún número tiene por sucesor el 1.*

Axioma cuarto. Dos números con el mismo sucesor son el mismo número. Esto es un desastre! Tal como tenemos el dibujo, con este nuevo axioma, el 1 y el 3 serían el mismo número (cosa que recuerda la santísima trinidad...). A partir de aquí, ya no tenemos otra salida que inventarnos un número infinito de números. Y por tanto, establecer alguna gramática como la base 10 para poderlos diferenciar, Figura 1.4.

Figura 1.4: *Axioma IV. Dos números con el mismo sucesor son el mismo número.*

A partir de estos cuatro axiomas, pues, no tenemos más remedio que inventarnos infinitos símbolos, creando así los números naturales.

Axioma quinto de Peano

Este último axioma sale un poco de la línea de los otros cuatro. Se trata precisamente del axioma de inducción. Parece que los primeros cuatro axiomas

sirvan para definir los números naturales, y el quinto para poder trabajar con ellos. Dice así:

Toda propiedad perteneciente al 1, y al sucesor de cualquier número que también tenga esa propiedad, pertenece a todos los números.

Este quinto axioma introduce el concepto de propiedad perteneciente a un número, concepto que no está incluido aún dentro de la teoría y de hecho, introduce una grieta en toda la teoría matemática. En cualquier caso, resulta bastante útil para poder sacar provecho, y por esta razón consideramos la inducción como una certeza incuestionable de la teoría de los números.

En la práctica, podremos utilizar la inducción para hacer demostraciones. Por inducción matemática sería fácil demostrar que las piezas del dominó caen como caen. A grandes rasgos, cuando se trate de demostrar una propiedad por inducción, nos encontraremos con expresiones genéricas con símbolos en su contenido. Es decir, fórmulas que contienen una hipótesis con un símbolo, n por ejemplo, para designar cualquier número. Entonces la demostración de la certeza del postulado se hará en dos pasos. En el primero sustituiremos n por el número 1 y alcanzaremos una verdad incuestionable. En el segundo paso, sustituiremos en la expresión inicial el símbolo n por $(n + 1)$ y haremos las transformaciones necesarias para conseguir aislar a partir de la nueva expresión el término con n que teníamos al inicio, para poder sustituirlo por lo que en aquella misma expresión inicial se postulaba. A esta sustitución le llamamos "por hipótesis de inducción", y una vez realizada, se tratará de alcanzar de nuevo una certeza incuestionable.

1.1.2 Sucesiones

En un conjunto de elementos de la misma naturaleza, elementos parecidos, podemos establecer una manera de identificar cada uno de ellos si los podemos reconocer por su posición en una secuencia ordenada. Entonces decimos también que tenemos un conjunto ordenado, y por tanto, tenemos un *índice* para identificar cada uno de los elementos. El índice es siempre un número natural. Considerar que el primero sea cero o uno no altera la teoría que se está formulando. No le daremos ninguna importancia.

Definición 1.1 Sucesión. *Llamamos sucesión a una secuencia ordenada de números reales.*

La manera más habitual de representar una sucesión es con los símbolos a_i, $i = 1, \ldots, n$, teniendo presente que se trata de números reales, $a_i \in \mathbb{R}$, $\forall i > 0$. También podemos encontrar expresiones como $a_1, a_2, \ldots, a_n$.

El número de términos de una sucesión puede ser finito, si n es un número

concreto, o infinito, cuando se define a_n, $\forall n > 0$. Cuando una sucesión es finita, se puede describir enumerando sus elementos (e.g. 1, 4, 9, 16), o también dando una expresión genérica, y acotando el número de índices a los que nos estamos refiriendo (e.g. $n^2, n = 1, \ldots, 4$). Cuando una sucesión es infinita, en cambio, si pretendemos describirla con una enumeración deberemos acabarla con puntos suspensivos (e.g. 1, 4, 9, 16, ...) que introduce cierta incertidumbre, ya que estamos suponiendo que de la observación de los primeros términos se pueden deducir los siguientes. De manera más rigurosa, podemos dar una expresión o fórmula genérica, y definirla para toda n mayor que cero (e.g. n^2, $\forall n > 0$). Entendemos entonces que n puede valer cualquier valor natural. También que podemos encontrar representaciones como $a_1, a_2, \ldots$.

En cualquier caso, siempre n es natural, $n \in \mathbb{N}$, que es equivalente a decir que es entero posistivo, $n \in \mathbb{Z}^+$.

Rápidamente podemos observar que hay dos tipos de sucesiones muy sencillos que llamamos progresiones:

- Las que cada número es el anterior más una constante.
- Las que cada número es el anterior multiplicado por una constante.

Un ejemplo de las primeras sería 1, 3, 5, 7, 9, 11 y 13. Un ejemplo de las segundas sería 2, 4, 8, 16 y 32. Es bien sabido que estas progresiones se llaman *aritméticas*, las primeras, y *geométricas*, las segundas. En las aritméticas, a la constante le llamamos *incremento* o *desplazamiento*. En las geométricas, *razón*.

La suma de los términos de una sucesión aritmética tiene una fórmula muy sencilla. De tan fácil que es, no es necesario recordarla. Podemos deducirla siempre que haga falta. Tan sólo se trata de darse cuenta de que el primero más el último suman igual que el segundo más el penúltimo, y que el tercero más el antepenúltimo. Esto es $1 + 13 = 3 + 11 = 5 + 9$. Y a partir de ahí, recordar que en total la suma es el primer número más el último multiplicado por la mitad de los números que haya. En lenguaje más formal, si notamos $a_i, i = 1, \ldots, n$ a los n términos de la sucesión, entonces la suma será $(a_1 + a_n)\ n/2$. En particular, la suma de esta es catorce multiplicado por tres y medio. En la ecuación (1.1) hay la expresión.

$$a_1 + a_2 + \cdots + a_n = \sum_{i=1}^{n} a_i = (a_1 + a_n)\ n/2 \tag{1.1}$$

Ahora supongamos que queremos sumar los términos de la progresión geométrica $a_i = 2^i$, $i = 1, \ldots, 5$. Es decir,

$$2 + 4 + 8 + 16 + 32$$

que suma 62. Hay una fórmula para hacer este tipo de sumas. Esta vez no es tan fácil de inventar, pero tampoco demasiado difícil de recordar. Llamando

$a_i, i = 1, \ldots, n$ a los n términos de la sucesión como antes, y denotando por r a la razón (en el ejemplo $r = 2$), entonces la fórmula para la suma de los n primeros términos será

$$\sum_{i=1}^{n} a_i = (a_1 - a_{n+1})/(1 - r). \tag{1.2}$$

Es decir, el primero menos el siguiente del último dividido por uno menos la razón, que en el ejemplo daría (2-64)/(1-2). Es curioso esto del siguiente del último, parece una contradicción... pero se entiende, ¿No?

Utilizamos la palabra *serie* para referirnos a la suma de los números de una sucesión cuando la sucesión es infinita, porque está bien claro que la suma de los términos de una sucesión finita es un número concreto, y por tanto no hay nada más que decir. En cambio, la suma de los términos de una sucesión infinita ya no queda tan claro. Es en este caso cuando el concepto de *serie* entra en escena. La denotamos, por ejemplo, con el símbolo S_n. Fijaos en que a partir de una serie que ya hemos definido como la suma de una sucesión de infinitos términos, nos inventamos una nueva sucesión. Un poco como un pez que se muerde la cola. O sea, partiendo de una progresión de infinitos números, $a_n, \forall n > 0$, nos inventamos el concepto de "suma de los términos de la progresión a_n". Para hacerlo, definimos S_n como la suma de los n primeros términos de la progresión inicial. $S_1 = a_1$, $S_2 = a_1 + a_2$, $S_3 = a_1 + a_2 + a_3$, y así sucesivamente hasta donde deseemos. De esta forma podemos concebir que S_n como tal, también es una sucesión. Decimos que S_n acumula a_n. La expresión genérica para la nueva sucesión S_n viene a ser

$$S_n \;=\; \sum_{i=1}^{n} a_i = a_1 + a_2 + \cdots + a_n,$$

y está claro que para que se pueda sumar para toda n, los a_i tendrán que ser decrecientes, o sea, la razón de la progresión deberá ser inferior a uno, $r < 1$.

Representación gráfica de una sucesión

No es difícil imaginar la manera de representar una sucesión en un plano cartesiano. En el eje horizontal colocaremos la variable independiente, el índice n, de manera equidistante. Y en vertical representaremos los valores de la sucesión sobre cada uno de los índices.

En la Figura 1.5 podemos ver la representación de la sucesión $a_n = 1/n$. En esa figura, los dos ejes tienen escalas diferentes. Las unidades en el eje vertical son mucho mayores que en el eje horizontal. Esto es así por comodidad, para que se vea mejor lo que se trata de comprender.

Observad que en el eje horizontal tenemos la variable independiente que por tanto podemos representar de manera equidistante entre los distintos valores.

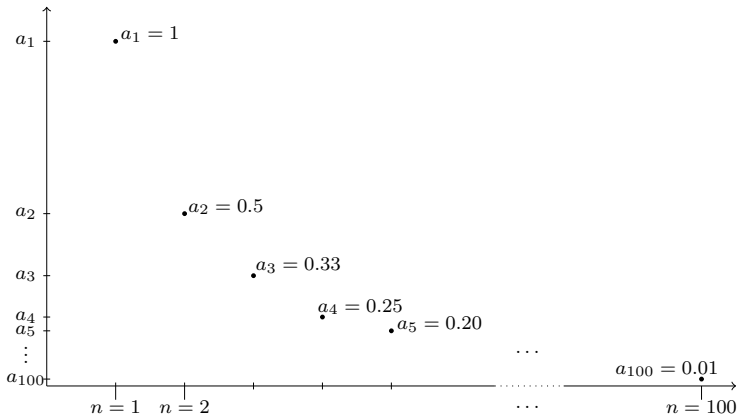


Figura 1.5: *Representación gráfica de la sucesión* $a_n = 1/n$.

Las distancias en vertical, en cambio, vienen dadas por la sucesión que representamos.

1.1.3 Límites

La paradoja dicotómica es una manera de plantear la paradoja de Aquiles y la Tortuga más fácil de entender. Paradoja quiere decir contradicción, pero de una forma más solemne. Esta paradoja se planteó hace unos dos mil quinientos años, en tiempos de la cultura griega. Se trata de responder a la siguiente cuestión:

Para poder recorrer una distancia, primero hay que recorrer la primera mitad. Pero para poder recorrer esa primera mitad, primero se deberá recorrer la primera mitad de esa mitad. Y como antes de recorrer la primera mitad de la primera mitad será necesario recorrer la primera mitad de la primera mitad de la primera mitad, y así hasta el infinito, entonces nunca se podrá recorrer ninguna distancia.

Parece pues una contradicción, porque resulta evidente que efectivamente sí que se pueden recorrer distancias. Así pues, ¿Cómo podemos explicárnoslo?

En pocas palabras, la respuesta se sintetiza en que se trata de una interpretación logarítmica del tiempo. Este es el error en el planteamiento de la paradoja. Dicho de otra manera, no es cierto que se tarde igual en recorrer la primera mitad que la primera mitad de la primera mitad. Aunque a la hora de decirlo, de verbalizarlo, parezca igual de largo (tardamos lo mismo para decir la primera *mitad* que la primera *mitad* de la primera mitad) la cantidad de tiempo a la que nos estamos refiriendo cada vez es menor. Es como si fuera tanto más pequeña, que se tardara más en sumar que la suma en crecer, y al final se tropezara contra su propio límite.

Parece que la matemática tuvo graves problemas para resolver esta paradoja. Por otra parte está documentada la suma de infinitos términos por Arquímedes hace dos mil tres cientos años. Pero, de hecho, pasaron como veinte siglos hasta introducirse el símbolo de infinito. Llegó con el cálculo diferencial, ya en tiempos del renacimiento. Los griegos de la época cuando se plantearon la paradoja, no se deberían a atrevir a suponer que la suma de infinitos números pudiera dar un número finito. Y efectivamente este es el caso en el qué nos encontramos. Y no es difícil de entender.

Numéricamente es sencillo. En resumidas cuentas, todas estas mitades son la sucesión $a_n = 1/2^n$, $\forall n > 0$. Y su suma es la serie geométrica

$$1/2 + 1/4 + 1/8 + ...$$

con razón $r = 1/2$.

Sin cometer ninguna aberración podemos suponer que el siguiente del último término será cero. O sea, el límite de la sucesión $a_n = 1/2^n$, cuando hacemos n infinitamente grande, es cero. Que también se puede decir $\lim_{n\to\infty} 1/2^n = 0$.

Entonces, utilizando la fórmula (1.2) de la página 10 tendríamos

$$(1/2 - 0)/(1 - 1/2),$$

que es igual a 1. Es decir, la suma de todas las mitades y las mitades de las mitades acaba sumando la distancia inicial que queríamos recorrer. Y por tanto, ahora ya nos cuadra que podamos hacerlo. Volviendo al lenguaje formal,

$$\lim_{n\to\infty} \sum_{i=1}^{n} 1/2^i = 1.$$

En síntesis, desde el inicio de esta sección, tan sólo hay un concepto que nos aporta alguna idea nueva. El concepto de límite. El punto más débil de todo el discurso. Vamos a robustecerlo. Supongamos por ejemplo que aparece algún escéptico incrédulo y nos dice que no se cree que el siguiente del último término sea cero. A nosotros nos corresponde satisfacerlo. Hay que poder convencerlo.

Definición 1.2 Límite. *Se dice que el límite de una sucesión infinita de números $a_n \in \mathbb{R}$, $\forall n > 0$, es L si la distancia entre los términos de la sucesión a_n y L puede ser reducida tanto como se desee aumentando n.*

Cuando una sucesión tiene un límite también se dice que la sucesión tiende al límite, o que la sucesión converge al límite. Asimismo, cuando una sucesión de números crece indefinidamente se dice que la sucesión tiende a infinito, o que diverge.

La misma definición en lenguaje formal podría escribirse[1]

$$\lim_{n\to\infty} a_n = L \Leftrightarrow \forall \epsilon > 0 \; \exists \; n^* \in \mathbb{Z}^+ \mid \forall \; n > n^* \rightarrow \; |a_n - L| < \epsilon.$$

Esta expresión se pronuncia diciendo *el límite de la sucesión a sub n es ele, si y sólo si para cualquier número épsilon mayor que cero existe un índice ene asterisco tal que los términos de la sucesión más allá de este índice están más cerca de ele que el número épsilon dado.*

En muchas disciplinas, épsilon acostumbra a denotar intervalos chiquititos. Conviene entender que el hecho de llamarle épsilon al primer número de la expresión es porque queremos insinuar que lo que estamos diciendo resultará interesante cuando este número sea pequeño. Muy pequeño. Tan pequeño como se desee. Y por muy pequeño que sea, siempre seremos capaces de encontrar un n^* eso es, no el primer término de la sucesión, ni el segundo, no. Ni el que hace mil ni el que hace un millón. Queremos decir el n^*_ésimo. A partir de ése, ya todos los números a_n para n's más grandes quedarán más cerca de L que la distancia acotada por ϵ.

Es decir, en palabras de la calle: "Si no te crees que $1/n$ tiende a cero cuando n crece, tu dime una distancia que, por pequeña que sea, yo te encontraré un número que, a partir de él todos los términos de la sucesión quedarán más cerca de cero que la distancia que tu me hayas dicho. Y eso, por muy pequeño que sea el número que tu me digas."

De totas formas, también debe haber quién no esté de acuerdo en que si dos puntos están tan cerca como queramos es que están en el mismo lugar... en fin, hay gente para todo.

Representación gráfica del límite

Se puede dar una interpretación gráfica de lo que se está diciendo, por si se quiere entender desde otro punto de vista.

Siguiendo con el ejemplo de antes, representamos cómo la sucesión $a_n = 1/n$ tiende a $L = 0$. Para esto, en la representación gráfica de la sucesión de la Figura 1.5 colocaremos el ϵ que nos diga el incrédulo en el eje vertical, a partir del límite al que la sucesión tiende.

Supongamos por ejemplo que el incrédulo quiere que le demos a partir de qué n los términos a_n estarán a una distancia de 0 inferior a 0.0005. Esto es, nos dicen que $\epsilon = 0.0005$. Reproduciendo el ejemplo de antes aumentado cerca de cero, tenemos lo que se representa en la Figura 1.6.

[1] Utilizamos la notación |a-b| para referirnos a la distancia entre a y b.

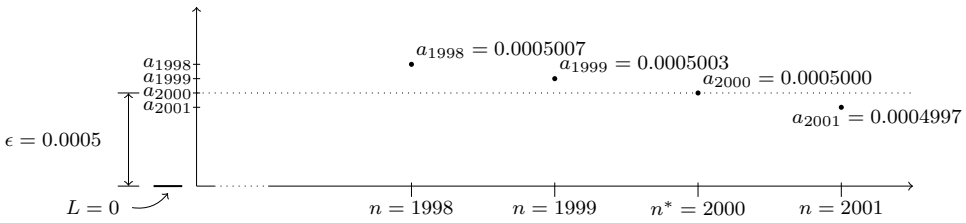


Figura 1.6: *Representación gráfica de la definición de límite para la sucesión* $a_n = 1/n$. *Nos piden aproximarnos al límite* $L = 0$, *a una distancia menor que* $\epsilon = 0.0005$. *La* n^* *correspondiente a este* ϵ *es* $n^* = 2000$.

Finalmente, quizás convenga reflexionar que tender no excluye estar. La sucesión $a_n = 37, \forall n > 0$, tiende a 37. Para las sucesiones constantes ocurre que el n^* no depende del ϵ. O sea, aún es más fácil, y para cualquier ϵ siempre se podrá responder que $n^* = 1$.

1.1.4 Clases de Equivalencia

Resumir es conocer. Sacar factor común es abstraer. La única forma de saber es estructurar el conocimiento de manera jerárquica, haciendo síntesis de los fenómenos. De todas las actividades que puede hacer nuestra mente, recordar, interpretar, calcular, percibir, identificar, etcétera, probablemente la más valuosa sea la capacidad de abstraer. Observar similitudes entre objetos o entre comportamientos y nombrarlas con alguna palabra. De hecho, cada palabra es una abstracción, y el lenguaje es uno de los tesoros más valuosos que tenemos.

Las clases de equivalencia son la única manera que tiene la matemática de realizar abstraciones. Por eso, es muy importante tener presente qué significan.

Probablemente el concepto más elemental de la matemática es el concepto de conjunto. Ligado al concepto de conjunto hay el de subconjunto. Éstos son conceptos que se supone que el lector tiene asumidos y por tanto no entraremos a definirlos. Aún así, quizás valga la pena recordar que los conjuntos no sólo se pueden definir por enumeración (o sea, diciendo cada uno de los elementos), sinó también como una propiedad o característica. Una característica define un conjunto. El conjunto de todos los objetos que tienen aquella característica. Así, podemos entender el conjunto de los animales de cuatro patas, sin nombrarlos uno por uno (cosa que además, en muchos casos sería imposible).

Bien, otra definición importante y muy cercana a la de conjunto es la de partición.

Definición 1.3 Partición. *Llamamos partición a una colección de subconjuntos de un conjunto grande si cada elemento del conjunto grande pertenece a uno, y*

tan sólo a uno, de los subconjuntos de la colección.

Es decir, cada elemento del conjunto grande pertenece a alguno de los subconjuntos de la partición, y además, ningún elemento pertenece a dos subconjuntos a la vez. En otras palabras, la intersección de cualquier pareja de conjuntos de la partición es vacía, y la unión de todos los conjuntos de la partición es igual al conjunto grande inicial.

De manera formal, si tenemos un conjunto C, y una colección de n subconjuntos de C, que llamamos C_i, para $i = 1, \ldots, n$, entonces

$$C_i, \quad i = 1, \ldots, n \text{ es una partición de } C, \text{ siendo } C_i \subset C \Leftrightarrow$$
$$C_j \cap C_k = \emptyset \qquad \forall j, k \in \{1, \ldots, n\}, j \neq k,$$
$$\text{y, además,} \qquad \bigcup_{i=1}^{n} C_i = C.$$

Una vez tenemos establecida una partición de los elementos de un conjunto, podemos decir que cada elemento de este conjunto grande inicial es de una clase C_i concreta. Es decir, de uno de los subconjuntos de la partición. Llamamos cardinal de la partición al número de clases de equivalencia, n.

Relaciones de equivalencia

Una *relación* es una frase que implica dos individuos de una población. Es lo mismo decir individuos de una población que elementos de un conjunto. O sea, no restringimos el concepto de individuos a personas. Una relación hace referencia a dos elementos de un conjunto y debe poder evaluarse como cierta o falsa para cada pareja de individuos, o de elementos. Las frases que se pueden evaluar como ciertas o falsas también se llaman *predicados*. Es decir, una relación es un predicado que involucra parejas de elementos de un conjunto. Para según qué pareja, la relación será cierta y diremos que esa pareja satisface la relación. Para otras, falsa. O sea, que otras parejas no la satisfarán. Por ejemplo "Ser del mismo color". Esta relación se puede definir sobre una población de camisas, por ejemplo. Diremos que dos camisas rojas satisfacen la relación. Si de las dos camisas de las que queremos decir algo son una verde y la otra amarilla, entonces diremos que no satisfacen la relación.

Ahora, además, calificamos las relaciones según tres adjetivos. Así diremos que una relación es...

- *Reflexiva* si un individuo consigo mismo la satisface.
- *Simétrica* si cuando un individuo la satisface con otro, el otro la satisface con el uno.
- *Transitiva* si cuando uno la satisface con otro y este otro con un tercero, el tercero la satisface con el primero.

Con todas estas definiciones concluimos diciendo que una relación es una relación de equivalencia si es reflexiva, simétrica y transitiva. Es decir, que cumple las tres propiedades.

Y ahora viene lo importante. Un secreto que sirve para ser un poco más sabios de una manera muy fácil.

Lema 1.1 *Una relación de equivalencia definida sobre una población de individuos establece una partición de esa población.*

Como ya se ha apuntado, a los subconjuntos que se crean a partir de la relación de equivalencia se les llama *clases*. Y se llaman *de equivalencia*, porque para lo que sea que queramos saber, sirve igual cualquier elemento de todos los de una misma clase. Ese elemento cualquiera de cada subconjunto es el *representante* de clase. Un representante de clase, pues, es equivalente a cualquier otro elemento de su misma clase.

Que una camisa es del mismo color que ella misma, está bien claro. Si una camisa es del mismo color que otra, esta otra es del mismo color que la primera. Y si una camisa es del mismo color que otra, y esa otra del mismo color que una tercera, entonces la primera camisa también es del mismo color que la tercera. Por tanto, la relación "Ser del mismo color" definida sobre la población de camisas, es una relación de equivalencia. Y por tanto define una partición. Eso significa que todas las camisas del mundo son de un color, o de otro. Y que no hay ninguna de dos colores a la vez...

Como siempre, la realidad es más complicada que la teoría, y está bien claro que hay camisas de dos colores. Y aún más difícil, camisas de colores extraños que no sabríamos cómo clasificar. Pero para el caso, para comprender el significado de lo que es una clase de equivalencia, este ejemplo ya resulta lo bastante útil.

Observad también un parámetro importante, la cardinalidad de la partición, n. Para el caso del ejemplo, sería el número de colores distintos, cosa que también es difícil de establecer.

Ejemplos más serios de particiones son, entre personas, "Haber nacido en el mismo país", "Tener la misma inicial del nombre", "Ser de la misma edad", o "Ser del mismo sexo". Es decir, todo el mundo ha nacido en algún país. Nadie ha nacido en dos países a la vez. Y si cogemos toda la gente que ha nacido en todos los países, cogemos toda la gente del mundo. Y las mismas cosas se pueden decir de cualquiera de los otros ejemplos.

Como consecuencia de estas últimas definiciones, somos más sabios. Podemos hablar de las propiedades de los elementos sin tener que enumerar todos los que las cumplen. Podemos decir que lo blanco es más claro que lo negro, y entendemos que todas las cosas blancas son más claras que todas las cosas

negras. Y eso, con todo el rigor, sin una pizca de incertidumbre.

Aún que haya quién lo disimule, canta que el tema de las propiedades reflexiva, simétrica y transitiva está intimamente ligado a la definición de los números naturales y el principio de inducción.

1.1.5 Funciones

Una función real de variable real representa una dependencia entre un número y otro. Podemos expresarla formalmente como

$$f : \underset{x}{\mathbb{R}} \to \underset{f(x)}{\mathbb{R}}$$

que se pronuncia *tenemos una función efe que va de erre en erre y a cada número real equis le da un número real efe de equis.*

Esta notación significa que tenemos un mecanismo que llamamos f, que dado un número perteneciente a los números reales, $x \in \mathbb{R}$, nos da otro número $f \in \mathbb{R}$ real. Y nada más. No nos da ninguna otra información.

No es el propósito de este texto profundizar sobre la naturaleza ni las propiedades que caracterizan las funciones. Se supone al lector un conocimiento bastante profundo sobre este concepto. En esta sección tan sólo se pretende aclarar el uso de los paréntesis en el ámbito del análisis matemático simbólico.

En primer lugar, hay que ser conscientes de que cuando se dice alguna cosa, sólo se dice lo que se dice. Y lo que no queda dicho, como no queda dicho, no tiene porqué alimentar ninguna sensación de incerteza. Eso ocurre especialmente con el uso de los paréntesis cuando hablamos de funciones de variables reales, o enteras. Que quede claro en adelante que $f(x)$ significa un número f que depende de x. Tan sólo utilizando la expresión $f(x)$, o con la expresión de más arriba, no estamos diciendo de qué manera depende. Y es que para lo que se quiere explicar, no es necesario decirlo.

Cuando una variable no depende de ninguna otra se llama variable independiente, y significa que podemos suponer que tiene el valor que nos convenga, si deseamos hacer tal suposición.

Dicho esto, nos disponemos a trabajar con las funciones como elementos de trabajo. Y más concretamente, nos disponemos a definir conjuntos de funciones. Estos conjuntos nos servirán finalmente para medir la eficiencia de los algoritmos. Es decir, es como si en lugar de decir que un listón mide 15 cm, dijerámos que el listón pertenece al conjunto de los listones que miden 15 cm. Son formas de hablar que quieren decir lo mismo.

Las funciones de las que hablaremos en todo momento no serán funciones de variable real, sinó de variable entera positiva. Es decir, nos dedicaremos a clasificar funciones del tipo

$$f : \underset{n}{\mathbb{N}} \to \underset{f(n)}{\mathbb{R}} .$$

1.2 Propósito del Algoritmia

El propósito del algoritmia es medir la complejidad de los problemas computacionales. O, dicho de otra manera, analizar la eficiencia de los algoritmos. Habitualmente, se habla de eficiencia a la hora de tratar todos los algoritmos en general, y se reserva la palabra complejidad cuando nos focalizamos en los más difíciles.

Así pues, empecemos.

Queremos tener una manera de medir la eficiencia de un algoritmo. Cuando se habla de eficiencia se hace referencia a dos recursos. El tiempo y el espacio. Hablaremos de eficiencia temporal para referirnos a la cantidad de tiempo que requiere un algoritmo para resolver el problema que resuelva. Y análogamente respeto a la eficiencia espacial. En cualquier caso haremos el estudio con la eficiencia temporal (todo lo que se diga, sin embargo, también vale para la eficiencia espacial). Siempre que no digamos a cuál nos estamos refiriendo, presupondremos que estamos hablando del tiempo requerido por el algoritmo en cuestión.

Para cuantificar la eficiencia lo haremos de la mejor manera que se nos ocurre. Y eso es a lo que nos dedicaremos el resto de este capítulo.

De entrada, veamos la definición formal de un algoritmo A.

Definición 1.4 Algoritmo. *Un algoritmo A es un procedimiento para resolver problemas de manera que transforma unos datos x, de un conjunto de posibles datos de entrada E, en una información y, de un conjunto de posibles salidas S, obtenida a partir de ellos.*

$$A : \underset{x}{E} \to \underset{y}{S}.$$

Es decir, dada una entrada de un conjunto de entradas posibles $x \in E$, el algoritmo A produce una salida de un conjunto de salidas posibles $y \in S$. Como ya se ha dicho, para hacer esto requiere un espacio y un tiempo. El concepto de procedimiento pretende dar a entender que el algoritmo hace uso de estos dos recursos básicos.

Daos cuenta que los conjuntos de entrada E y el de salida S pueden ser muy grandes. Por ejemplo, si tenemos un algoritmo para sumar dos números, el conjunto de entrada es el conjunto de todas las parejas posibles de números, y el de salida, el conjunto de todos los números. Así pues, si para un algoritmo tan sencillo como una simple suma estos conjuntos ya son infinitamente grandes, imaginaos cuando no se trata de una suma sinó de ordenar una lista, o de acciones más complicadas.

Nuestra intención es calcular la eficiencia del algoritmo A. Dicho de otra forma, calcular cuánto tiempo tarda a dar la respuesta una vez le hemos dado los datos.

Nos encontramos con un inconveniente que lo complica todo.

En seguida observamos que el tiempo que se tarda en hacer una suma de dos números de una cifra es inferior al de cuando los dos números tienen treinta mil cifras.

Este es un obstáculo grave que casi provoca que dejemos correr nuestro propósito, y tiremos este libro por la ventana. Que nos echemos atrás a la hora de poder medir la dificultad de resolver un problema. De hecho, nos gustaría mucho que el tiempo que tarda un algoritmo no dependiera de los datos de entrada, pero ocurre. Nosotros, testarudos, no nos rendimos, y nos preguntamos con coraje: "A ver, ¿Cuántas entradas posibles puede tener el algoritmo?". De hecho si podiéramos probar el algoritmo con todas las entradas posibles la cosa sería sencilla. Podríamos definir la eficiencia del algoritmo como la media de tiempo que tarda entre todas sus entradas. Pero está claro, no podemos. Entonces nos desanimamos porque la respuesta es: "Muchas. Demasiadas entradas para poder considerarlas todas". Bien, seguiremos hablando de ello.

Dado un conjunto de datos de entrada concreto, diremos que tenemos una *instancia* del problema que resuelve el algoritmo en cuestión. Es decir, utilizaremos el término instancia para lo que normalmente se le llama problema. Nosotros utilizamos la palabra problema para hablar del problema genérico, sin especificar ningún conjunto de datos de entrada. Y de momento, mantenemos los conceptos de problema y algoritmo asociados uno a uno.

1.2.1 Herramientas que utilizaremos

De cara a conseguir poder medir el tiempo que tarda un algoritmo a dar una solución a partir de unos datos, es decir, de cara a analizar la eficiencia de un algoritmo, vamos paso a paso. Primero necesitamos definir un par de herramientas que nos resultarán útiles para nuestro propósito: El tamaño de los datos, y el tiempo de un algoritmo para una entrada de tamaño n. Como se ve, decir los datos, los datos de entrada, o la entrada, es lo mismo.

Tamaño de los datos

Ya que el conjunto de datos de entrada es demasiado grande, particionémoslo. Dividamos el conjunto de todas las entradas posibles $x \in E$ en grupos con alguna relación de equivalencia. Es una gran idea que nos permitirá continuar nuestro estudio. Nos inventamos lo que llamaremos *tamaño* de los datos. El tamaño de una entrada será un número entero positivo. La manera como asignaremos un tamaño a cada entrada posible ya lo veremos más adelante. De momento suponemos que disponemos de alguna forma de decir que una entrada concreta $x \in E$, tiene un tamaño $n \in \mathbb{N}$. Formalmente, para el tamaño utilizaremos la notación de barras verticals, como con las cardinalidades de los conjuntos. De hecho, hay una relación bastante clara.

$$| \, . \, | : \underset{x}{E} \to \underset{n=|x|}{\mathbb{N}} \tag{1.3}$$

El hecho de poner un punto entre las dos barras en la expresión (1.3) pretende indicar que usaremos esa función con notación infija. Es decir, en lugar de denotarla como *tamaño(x)*, la denotaremos $|x|$. Así pues, el tamaño es una función que para cada elemento del conjunto de entradas posibles a un algoritmo nos da un número natural que nos gusta llamarle n.

Y de la misma manera que la relación de equivalencia "Tener la misma edad" particiona la población de todas las personas del mundo, la relación de equivalencia "Tenir el mismo tamaño" particiona todas las posibles entradas de cada algoritmo. Porque está bien claro que una entrada medirá el mismo tamaño que ella misma. También ocurrirá que si una entrada mide igual que otra, entonces la otra mide igual que la una. Y también, que si una entrada mide igual que otra y esta que una tercera, la última mide igual que la primera. O sea, "Tener el mismo tamaño" es una relación de equivalencia.

Bien, de momento hemos conseguido que para poder tratar todas las entradas posibles sólo se requiera poder tratar las entradas de todos los tamaños posibles. Ahora, además, cometeremos un abuso. Diremos, aunque sea mentira, que el tiempo que tarda un algoritmo a dar la solución a un problema con una entrada de tamaño n sólo depende de este tamaño, sin tener en cuenta un hecho muy importante: Para entradas del mismo tamaño el algoritmo puede tardar tiempos muy diferentes.

Tiempo de un algoritmo para una entrada de tamaño n

Huyendo hacia delante, para el tiempo que tarda el algoritmo $A : E \to S$ con una entrada $x \in E$ de tamaño $|x| = n$ a dar una salida $y \in S$, utilizamos la notación $T_A(n)$. Así, con la T mayúscula. Ponemos la T mayúscula para recordar que en el fondo, y por mucho que lo deseemos, $T_A(n)$ no es un número. Es necesario tener presente que para diferentes entradas del mismo tamaño el algoritmo A puede tardar tiempos diferentes. Calcular $T_A(n)$ es la finalidad de nuestro estudio. Es aquello que nos proponemos. Y está bien claro que en la notación $T_A(n)$ no aparece la entrada x. Sólo su tamaño, $|x| = n$.

Definimos el tiempo de un algoritmo para una entrada como si sólo dependiera del tamaño de esta entrada.

La única cosa que podemos hacer llegados a este punto es recorrer a la estadística. Precisamente la estadística es especialista en resumir colecciones de números en un sólo número, y éste es el inconveniente que tenemos en este momento.

Definimos el *caso peor* de un algoritmo como la entrada con la que más tarda a responder de entre todas las entradas de un mismo tamaño. Así, habremos conseguido tener en un sólo número (ahora sí que será un sólo número) una información concerniente a todas las entradas de aquel tamaño. Decimos caso peor porque, está bien claro, cuando más rápido sea el algoritmo más eficiente será.

Y por fin, dado un algoritmo $A : E \to S$, definimos el tiempo que tarda el algoritmo A para una entrada, o caso, $x \in E$ como el máximo de todas las entradas con el mismo tamaño (el tiempo de la respuesta que más tarda).

$$t_A(n) = \max_{x \in E} \{T_A(n) \mid |x| = n\}$$

De todas formas, como no tenemos ninguna manera sistemática de caracterizar el caso peor, cuando hablemos del tiempo de un algoritmo utilizaremos $T_A(n)$ y no $t_A(n)$.

Los diferentes casos significa las diferentes entradas x por un mismo n. En otras palabras, cuando hablamos de casos medio o peor nos estamos refiriendo a los valores concretos de las diferentes partes que compongan la entrada x.

Es conveniente hacer una asociación mental entre los términos "caso peor" y "valores". Siempre que hablemos del caso peor, hablaremos de los valores de la entrada peor. Y para el caso medio o el mejor, también, claro. Cuando de un algoritmo nos pregunten por el caso peor, la respuesta ha de ser que el caso peor se produce cuando los valores de la entrada cumplan ciertas condiciones.

Por otra parte y aunque no se le dé tanta presencia, el análisis de la eficiencia espacial puede ser realizado de manera paralela al temporal. Probablemente, como la cantidad de espacio que requiere un algoritmo puede hacerse disponible siempre que la economía lo permita (cosa que no sucede con el tiempo), la importancia que se le presta a la eficiencia espacial queda siempre una poco más al margen que la temporal. En cualquier caso, es necesario comprender que tanto la notación asintótica como todo el resto de técnicas que se utilizan para la eficiencia temporal, también sirven para el cálculo de la espacial.

Las unidades de tiempo que utilizaremos tendrán poca importancia. Para el lector inquieto, la respuesta podría ser operaciones elementales. El tiempo de una suma, o el tiempo de una asignación pueden considerarse como las unidades fundamentales indivisibles. Más adelante se irá viendo que estos tiempos son infinitamente chicos y sólo consideramos su existencia en cuanto son multiplicados por números muy grandes. Es decir, las unidades de tiempo son muy pequeñas, pero no cero ya que si las multiplicamos por números grandes, el producto es diferente de cero.

En síntesis, de esta sección es importante recordar que un algoritmo nos transforma datos de entrada en datos de salida en un tiempo que lo definimos no en función de la entrada sinó tan sólo del tamaño de esa entrada, pero está bien definido en el sentido de que para cada entrada podemos obtener un número concreto que representa este tiempo.

También hay que tener en cuenta que hay algunas cosas que nos hemos dejado en el tintero. No ha quedado claro como calculamos el tamaño de una entrada. Además, teniendo en cuenta que este tiempo que asociamos a un algoritmo para una entrada de un tamaño dado tiene que ver con el peor de los casos para todas las entradas de ese tamaño, tampoco ha quedado nada claro cómo reconocer cuál es ese caso peor al que nos estamos refiriendo (de hecho esto dependerá de cada problema).

Lo que sí que queda claro, en cambio, es que el peor caso es el que tarda más. O uno cualquiera suponiendo que haya varios casos máximos. Es decir, entre todos los casos que tardan el máximo tiempo para las entradas de un tamaño dado, cualquiera de ellos se puede considerar el caso peor, ya que todos esos máximos serán equivalentes.

Un error muy grave que muchos estudiantes cometen es recurrir al tamaño de los datos cuando se trata de identificar el caso peor. Eso significa no entender nada. No se puede decir nunca en la vida que para tal o cual algoritmo el caso mejor es cuando n sea 1, y entonces tardará lo que sea. No. Esto no es así. Para identificar los casos peor, medio, o mejor, jamás podemos recurrir al tamaño de los datos. Hay que identificar los casos para un tamaño cualquiera, para todos los tamaños. Y hay que utilizar el concepto de valor para encaminar el discurso. Los valores de la entrada para el caso peor cumplen alguna propiedad y es ésta, la que identifica el caso.

1.3 Conjuntos de la Notación Asintótica

1.3.1 Introducción

En el principio de este libro hay una frase bien extraña: *qué es a cómo lo que cuántos es a cuáles*. Lo que pretende transmitir es el papel que juegan los conceptos que resumen muy brevemente una gran cantidad de información, que podríamos llamar lcqrmbugcdi.

Por eso, fijaos en el hecho de que el término *qué* resume en una o dos palabras todo un universo que hay detrás del *cómo*. Asimismo, y quizás de una manera más clara *cuántos* sintetiza en una palabra todo el despliegue que insinúa *cuáles*.

La notación asintótica, en definitiva, compacta en una expresión muy breve, mucha información del algoritmo que mide.

Saliendo un poco del hilo argumental del algoritmia hacemos una breve incursión en otro tema. En la frase hay dos referencias: una al tiempo (*qué, cómo*), y la otra al espacio (*cuántos, cuáles*). Sigue ahora una disertación. Hacemos un inciso en el tema de la dualidad espacio tiempo. Se aconseja al lector más pragmático interesado exclusivamente en la notación asintótica que pase directamente a la siguiente sección.

El juego del Espacio/Tiempo

Démonos cuenta de las analogías entre todo y siempre, sustantivo y verbo, objeto y acción, adjetivos y adverbios, poesía y novela, lo lírico y lo épico, etcétera.

Hay un juego muy interesante para hacer gimnasia intelectual. Se llama el juego de la dualidad espacio tiempo, y está basado en la teoría de conjuntos

Hay dos jugadores: el entrenador y el corredor.

Tiene dos fases, la de los sustantivos y la de los verbos. Primero, el entrenador le explica al corredor que se trata de utilizar la estructura verbal siguiente:

Todo A es B pero no todo B es A.

Es decir, el entrenador dirá parejas de palabras (sustantivos) en cualquier orden, y a partir de su semántica, el corredor tiene que ser capaz de formular la expresión con esta estructura (el secreto del entrenador es que las parejas de palabras que propone cumplen una regla: un sustantivo es subconjunto del otro).

Por ejemplo el entrenador dice: "Gato y Animal '. Entonces el corredor debe responder: "Todo gato es animal pero no todo animal es gato". Entonces, el entrenador dice "Mueble y silla". Y el corredor debe decir: "Toda silla es un mueble, pero no todo mueble es una silla"... y así hasta que se cansen, o hasta que el corredor ya no falle nunca.

Entonces, cuando el corredor ha superado la fase de los sustantivos, se pasa a la fase de los verbos.

La estructura que tiene que seguir en esta segunda fase es:

Siempre que se (hace) A, se (hace) B pero no siempre que se (hace) B se (hace) A.

En principio, lo ideal se usar el presente de subjuntivo, aunque que sin ser conscientes de ello, ya sale así.

Por ejemplo, el entrenador dice: "Tocar y Rascar". Y el corredor responde: "Siempre que se rasca se toca, pero no siempre que se toca se rasca". Después, el entrenador: "Recordar y Pensar". Y el otro: "Siempre que se recuerda se piensa pero no siempre que se piensa se recuerda", "Comer e Ingerir"... y así hasta que el corredor ya no se equivoque nunca. De hecho, no hace falta que sean tan sólo parejas de palabras. También pueden ser parejas de expresiones ("Hacer palmas" y "Hacer ruido', "Dar besos" y "Amar",...).

Finalmente, la victoria del corredor se produce en el momento en que es él quien es capaz de asumir el papel de entrenador, y entonces el juego ya se ha acabado.

Volviendo a la frase inicial, casi todo el mundo debería estar de acuerdo en que dada una cantidad de objetos cualesquiera, si sabemos cuáles son, sabremos cuántos son. Lo mismo también pasa pero no con una colección de objetos sinó con una colección de procesos, es decir, ya no hablamos de sustantivos sinó de acciones. Entonces, si sabemos cómo se hace una cosa, también sabremos qué se hace.

En definitiva, desde los dos puntos de vista se está explicando que la primera palabra (*qué* o *cuántos*) es un resumen muy sintético de la segunda (*cómo* o *cuáles*). Són resúmenes con pérdidas de información. Resúmenes que se quedan con una mínima parte de la información que resumen.

Bien, toda esta introducción es debida a que conviene comprender que medir los objetos es quizás la manera más concisa, más compacta o más resumida de conocerlos. Es decir, la notación asintótica que nos ha de servir para medir la complejidad de los algoritmos, también nos servirá, en definitiva, para conocerlos.

1.4 Los conjuntos de funciones O, Θ, y Ω

Dicho de forma sencilla, una función que depende de un número natural $f = f(n)$ pertenece a $O(n)$ si cuando aumentamos el valor de n tanto como queramos, la función crece más poquito a poco o igual (es decir, no más de prisa) que n. Hay que tener en cuenta, sin embargo, que n, por sí misma, también es una función de n. Esto así explicado, pues, es una simplificación.

Cuando definimos $O(n)$, estamos simplificando la explicación ya que la definición correcta no tiene por qué tomar de referencia la misma n, sinó cualquier función de n.

Ahora bien, para establecer la definición sí que lo hacemos con todo rigor.

Definición 1.5 *$O(g(n))$ "o de g de n". Dadas $f = f(n)$ y $g = g(n)$, se dice que la función f pertenece al conjunto $O(g)$ si cuando n se hace muy grande, f no crece más rapidamente que g.*

$$f \in O(g) \Leftrightarrow \lim_{n\to\infty} f/g < \infty.$$

Definición 1.6 *$\Theta(g(n))$ "zeta de g de n". Dadas $f = f(n)$ y $g = g(n)$, se dice que la función f pertenece al conjunto $\Theta(g)$ si cuando n se hace muy grande, f crece como g.*

$$f \in \Theta(g) \Leftrightarrow \lim_{n\to\infty} f/g = c, \quad c \neq 0, c \in \mathbb{R}.$$

Definición 1.7 *$\Omega(g(n))$ "omega de g de n". Dadas $f = f(n)$ y $g = g(n)$, se dice que la función f pertenece al conjunto $\Omega(g)$ si cuando n se hace muy grande, f no crece más lentamente que g.*

$$f \in \Omega(g) \Leftrightarrow \lim_{n\to\infty} f/g > 0$$

Para cada g que nos imaginemos, $O(g)$ es un conjunto de funciones. $\Theta(n)$, $\Theta(n^2)$, $\Omega(\sqrt{n})$, $O(1)$... Fijaos en que O, Θ, o Ω como tales, sin paréntesis, son coleciones de conjuntos. El papel que hacen las diferentes funciones que puede representar $g(n)$ es de referencia en la notación asintótica. Si recordais un poco como se calculan los límites, está bien claro que el conjunto $\Theta(n)$ es el mismo conjunto que $\Theta(2n)$.

Observad también que

$$\Theta(f(n)) = O(f(n)) \cap \Omega(f(n)), \quad \forall f \in \mathbb{L}.$$

siendo $\mathbb{L}$ el espacio de todas las funciones de variable real.

Una cosa que se podría objetar es que la definición de $\Theta(g(n))$ no es correcta, ya que si $g = g(n)$ y $f = f(n)$ entonces para decir que una crece como la otra el límite debería ser 1, y no cualquier otra constante. Bien, respeto a este punto es conveniente recordar que estamos haciendo una definición, y la hacemos como nos interesa hacerla. Para ser una poco más convincente, mantengamos presente que cuando se dice que f crece como g quiere decir con un grado de libertad.

Permitimos así que los crecimientos, cuando se mantienen separados por una constante, sean clasificados como equivalentes. Y ¿Por qué?. Pues muy sencillo: Para absorver el error que cometemos al no definir las unidades con las que medimos n. Es decir, si el límite de f entre g es una constante, entonces variando las unidades con las qué medimos n podemos hacer que sea 1. Esto nos lleva, sin embargo, a un otra duda, ya que la réplica automática sería "Hombre, sean cuáles sean las unidades, deberían de ser las mismas para f que para g". Réplica que efectivamente es correcta. La respuesta entonces ya es más complicada, e iremos hablando de ello. Que quede claro ahora, en cualquier caso, que existe esta c, y que la llamamos *constante oculta* o *constante escondida* de la notación asintótica.

La constante oculta en la definición de $\Theta(g(n))$

La constante oculta c de la definición de $\Theta(g(n))$ se llama así porque cuando decimos que una función $f = f(n)$ pertenece a $\Theta(g(n))$ ignoramos la constante deliberadamente. Esto implica que podemos decir que dos algoritmos tardan igual cuando realmente uno tarda el doble que el otro, o el triple, o lo mismo que el otro multiplicado por mil. Estamos introduciendo pues un grado de imprecisión considerable.

Recordad que cuando definíamos el tiempo de un algoritmo A a dar una salida para una entrada x de tamaño n nos resultaba una definición, $T_A(n)$, de una cosa que no era un número. En rigor, $T_A(n)$ es una distribución de probabilidad sobre la variable aleatoria $x \in E$, de entre todas las de tamaño $|x| = n$. La única manera posible de ahorrarnos el estudio de estas distribuciones de probabilidad es dando este grado de libertad a la definición de $\Theta(g(n))$. Es decir, hacemos la vista gorda para poder verlo todo.

En otras palabras, si pretendemos diferenciar entre un algoritmo que tarda un tiempo a responder a una entrada y otro que tarda el doble de tiempo con la misma entrada, entonces no podremos utilizar el mismo instrumento de medida para un tercer algoritmo que tarde una cantidad de tiempo exponencial con la misma entrada.

El problema con que nos topamos es un problema de escala. No se puede hacer un mapa de Europa y que aparezca un campo de fútbol a escala. De hecho, Valencia es más del doble de grande que Burgos, y en cambio en un mapa de España sale igualmente un punto para indicar cualquiera de las dos

ciudades.

Con todo, no se quiere decir que no seamos capaces de diferenciar entre un algoritmo que tarda tanto y otro que tarda el doble. Efectivamente sí que podremos diferenciar cuando nos interese hacerlo. Pero entonces ya no estaremos utilizando la notación asintótica. Entonces ya estaremos hablando exclusivamente de la constante oculta.

Como consecuencias de no tener presente esta constante cuando se utiliza la notación asintótica podemos observar que:

- Consideraremos que se tarda igual en hacer una suma que dos sumas o que cien mil sumas. De hecho, cualquier número concreto de sumas consideraremos que tarda como una sola suma. Lo que sí que tardará más que una suma será n sumas, siendo n el tamaño de los datos de entrada. Y más que n, pues $n\ log(n)$, o n^2.

- El tiempo de cálculo de una operación elemental se puede considerar infinitamente chico.

- Las unidades con qué medimos los datos no tienen ningún impacto en el análisis de eficiencia.

Usos habituales de los tres conjuntos

Tanto para la eficiencia temporal como para la espacial utilizaremos los conjuntos Θ siempre que podamos, ya que es lo que más información nos da.

Por otra parte, cuando hablemos de eficiencia temporal de un algoritmo y no seamos capaces de acotar con suficiente precisión su tiempo, entonces acostumbraremos a utilizar los conjuntos O, ya que, siempre nos va interesar cuanto tarda el algoritmo como máximo. Si no estamos seguros de si un algoritmo es $\Theta(n)$ o $O(n)$, entonces diciendo $O(n)$ nos curamos en salud, aunque la respuesta es más imprecisa y por tanto menos correcta. Recordad que si se da una eficiencia con los conjuntos O lo que se está dando es una cota superior del tiempo que tarda el algoritmo. Por tanto, es cierto que los algoritmos que se usan actualmente son todos $O(n^n)$, aunque esto no nos aporte demasiada información.

Y finalmente, cuando se trate de eficiencia espacial, nos pueden interesar ambos conjuntos, tanto Ω como O. Probablemente utilizaremos más a menudo $\Omega(g(n))$ ya que, hablando de espacio, la preocupación más habitual será saber cuanto espacio necesita el algoritmo como mínimo.

Cálculos que utilizan los conjuntos de funciones con el símbolo de igualdad inclusivo

A pesar de que estemos acostumbrados a utilizar el símbolo de igualdad ($=$) para expresar relaciones simétricas (es decir, normalmente si $a = b$, entonces $b = a$), no cuesta entender que también se pueda utilizar, en lugar de como *igual*, como *pertenece* ($\in$).

Es lo mismo decir que $f(n) = \Theta(n)$ que que $f(n) \in \Theta(n)$. Esto lo convenimos así porque como se verá seguidamente, a menudo utilizaremos los conjuntos como términos en expresiones formales. Es decir, cálculos correctos pueden tener la forma,

$$T(n) = n^2 * O(n^3) = O(n^5).$$

Lo que en ningún caso es aceptable es poner el conjunto en la izquierda del signo de igualdad cuando en la derecha no aparezca ningún otro conjunto. O sea, tiene sentido decir que $f(n) = \Theta(n)$, o que $O(f) = O(g)$. Lo inadmisible se escribir expresiones como $O(n) = f(n)$.

Propiedades de los conjuntos de funciones

No es difícil imaginarse que las propiedades de los conjuntos O, Θ y Ω se desprenden directamente de su definición, y por tanto, de la definición de límite. Dadas $f = f(n)$, $g = g(n)$ y $h = h(n)$, y también $a \in \mathbb{R}$ tenemos:

- $\Theta(f(n))$ define una relación de equivalencia, ya que se cumple que
  - $f \in \Theta(f)$.
  - $f \in \Theta(g) \rightarrow g \in \Theta(f)$.
  - $f \in \Theta(g) \wedge g \in \Theta(h) \rightarrow f \in \Theta(h)$.
- $\Theta(an) = a\Theta(n)$. También $O(an) = aO(n)$, y $\Omega(an) = a\Omega(n)$.
- $\Theta(a + n) = \Theta(n)$. También $O(a + n) = O(n)$, y $\Omega(a + n) = \Omega(n)$.
- *(regla de la suma)* $\Theta(f+g) = \Theta(max(f, g)$. También $O(f+g) = O(max(f, g)$, y $\Omega(f + g) = \Omega(max(f, g))$

Para el caso de la regla de la suma, cuando la eficiencia de un algoritmo viene dada por el máximo entre dos partes del algoritmo, entonces se dice que la eficiencia *viene dominada* por la parte que dé ese máximo. Por otro lado, observad que ni $O(n)$ ni $\Omega(n)$ definen relaciones de equivalencia, ya que no son simétricas.

- $O(n) = O(n^2)$, pero $O(n^2) \neq O(n)$.

- $\Omega(n^2) = \Omega(n)$, pero $\Omega(n) \neq \Omega(n^2)$.

Funciones de referencia habituales

Como se ha dicho en la definición de los conjuntos de funciones de la página 25, para las funciones $g(n)$ de aquellas definiciones tendremos unas cuantas candidatas que utilizaremos habitualmente. En la Sección 1.4 anterior, ya se ha visto que Θ define una relación de equivalencia en el espacio de las funciones reales $\mathbb{L}$. Eso significa que particiona ese espacio, ya que para esto nos interesa que sea una relación de equivalencia. Lo que haremos, pues, es designar una función $f(n)$ como representante de cada clase. Luego, para cualquier algoritmo tendremos que el tiempo que tarda a obtener una salida a partir de una entrada de tamaño n, $T_A(n)$, pertenecerá a alguna $\Theta(f(n))$. Así, habremos analizado la eficiencia del algoritmo, y conseguido nuestro propósito.

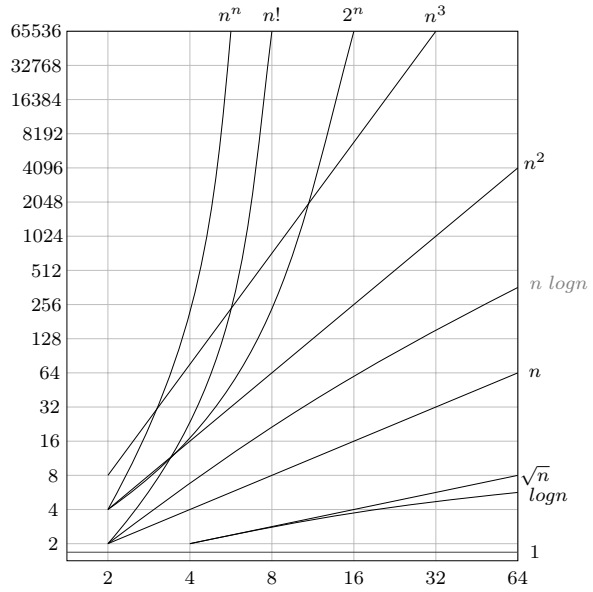


Figura 1.7: *Representación gráfica de las funciones básicas que utilizaremos como referencia en la notación asintótica.*

En la Figura 1.7 podemos ver las funciones de referencia, o representantes de clases de crecimiento.

Se debe observar, en primer lugar, que los ejes de coordenadas son logarítmicos. O sea, que a intervalos equidistantes tenemos representados valores multiplicativos. Así, podemos dibujar en un espacio reducido grandes intervalos numéricos. De rebote, pero, hay curbas que toman aspecto de línea recta, como el caso de n^2.

Fijaos que más allá de $n = 16$, ya no hay ningún cruce, de manera que cuando no tengamos la gráfica delante, tan sólo evaluando cada función en el punto $n = 16$ ya las podremos ordenar.

Les diez funciones dibujadas en la Figura 1.7 muestran como puede aumentar el tiempo de respuesta de los algoritmos para entradas de tamaño n, representada en el eje horizontal, siempre que las utilizemos como argumento de $\Theta(f(n))$. Cuando las utilizemos como argumento de $O(f(n))$ significan cotas superiores para los tiempos de los algoritmos. Y para $\Omega(f(n))$, inferiores (que si hablamos de tiempos, no tienen demasiado interés).

Vemos que el crecimiento más lento es $f(n) = \Theta(1)$ y por tanto los algoritmos más rápidos. Éstos serán algoritmos que no se miren la entrada. El "Hellow world", o un algoritmo para sumar dos más dos son ejemplos bien claros.

Hablemos del logaritmo. Etimológicamente, *log* viene del griego y significa algo así como indicador, estudio o conocimiento. De *log* viene el sufijo *logía* tan famoso, de biología, ecología o geología. Y también viene del griego *aritmo* que significa número. O sea, que logaritmo quiere decir indicador del número. De hecho, si pensamos que el logaritmo de un número es el número de cifras que tiene el número no nos equivocamos mucho. Vale la pena, pues, recordarlo así.

El logaritmo de un número es la cantidad de cifras que tiene.

Enriqueciendo la sentencia, el logaritmo de un número en una base es la cantidad de cifras que tiene en aquella base. Si alguna vez se nos pide calcular un logaritmo en alguna base, por ejemplo $log_4 5$, comenzamos pensando *igual a* x. Nos agarramos a la incógnita x como a un tablón de náufrago. O sea, si nos preguntan, ¿Cuál és el logaritmo en base 4 de 5?. En seguida, pensamos... A ver, si logaritmo en base 4 de 5 es x..., y entonces quizás pueda servir de ayuda imaginar un péndulo con el extremo superior clavado sobre el signo de igual, y del que cuelga la base del logaritmo, el 4. Entonces, el péndulo cae y oscila de manera que empuja el valor a calcular, x, hacia el exponente.

$$log_4 5 = x \rightarrow 5 = 4^x$$

Como el logaritmo de un número en una base es el número de cifras que tiene el número en aquella base, en la Figura 1.8 se puede observar una manera bien fácil de hacerse una idea de lo lento que es un crecimiento logarítmico. Fijaos que escribiendo todos los números, uno bajo el otro, el perfil que se dibuja coincide con el crecimiento de la función $log(n)$ (en la figura se ha girado el papel).

Por definición de la notación asintótica, la base del logaritmo no importa.

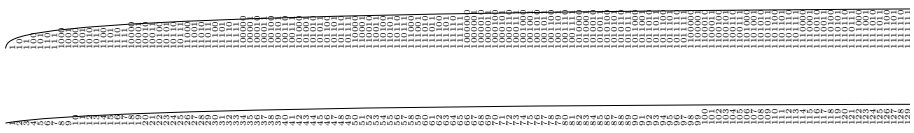


Figura 1.8: *Forma sencilla de recordar la lentitud en el crecimiento de las funciones logarítmicas: Para dibujar la curba correspondiente al logaritmo en una base, escribimos los números en esta base uno bajo el otro. Entonces el perfil que dibujamos se corresponde con la forma del logaritmo. En la parte superior, logaritmo en base 2. En la inferior, logaritmo decimal.*

Recordad que cuando escribíamos un número en hexadecimal (utilizando pues las cifras 0,...,9,A,...,F) lo hacíamos porque la traducción a binario era inmediata: Cada cifra hexadecimal son exactamente cuatro cifras en binario. Esto es, dos logaritmos de x en diferentes bases difieren en un factor constante para cualquier x. Es por esto que en la notación asintótica no impacta la base de los logaritmos. Porque ese factor queda absorvido en la constante oculta. En otras palabras, $log_b(x) = log_a(x)log_b(a)$, que para el caso del hexadecimal a binario, la constante $log_b(a)$ es $log_2(16) = 4$.

Este conocimiento también debería ser útil para poder acotar a ojo números dados en base 2. Ahora, nos planteamos la siguiente cuestión:

¿Cuántas cifras tiene en decimal el número 2^{32}?. Es decir, un número que en binario tendría 32 cifras.

La respuesta, a partir de lo dicho anteriormente es fácil de calcular. Primero pensemos en cuál es el $log_2(10)$ que viene a ser 3 y pico, ya que 2^3 da 8. Entonces, hay que dividir 32 entre 3 y pico, que da diez, más o menos. O sea que el número 2^{32} tiene unas 10 cifras en decimal aproximadamente. Es mayor que mil millones y menor que diez mil millones.

Otro ejemplo. Imaginaos que tenemos un problema consistente en guardar un cordel de zapato en el bolsillo. Nos hace falta un algoritmo para plegar el cordel. Tomamos como tamaño de la instancia, n, la longitud del cordel, en centímetros o cualquier otra unidad.

Básicamente lo podemos resolver de dos formas. La primera sería poco eficiente. Se trata de enrollar el cordel alrededor de la mano, tal como se muestra en la Figura 1.9.

Fijaos bien que el número de vueltas que hace el cordel a la mano sería el doble si el cordel fuera el doble de largo. Eso es $\Theta(n)$. Y la analogía va más allá. Podríamos afirmar que la holgura que queda entre el haz y la mano, el radio de la circunferencia, define una constante oculta concreta. Si el cordel, al enrollarse en la mano se pegara y no dejara espacio, entonces la constante oculta sería más alta, y por tanto el algoritmo menos eficiente. Daríamos más

Figura 1.9: *Algoritmo polinómico para guardar un cordel en el bolsillo.*

vueltas para guardar el mismo cordel, pero en cualquier caso, la relación entre la longitud y el número de vueltas sería lineal.

Conviene también pensar, desde un ángulo más filosófico, que el hecho de ser $\Theta(n)$ está relacionado con que la mano toca cada uno de los centímetros del cordel. En este caso es tocar, pero en general, también podríamos decir tratar. En el procedimiento, la mano trata cada centímetro del cordel. Como conclusión, cualquier algoritmo que lea o escriba los datos que trata será, como mínimo, $\Theta(n)$, puesto que al leer o escribir se tratan cada uno de esos datos.

Para la ilustración del segundo algoritmo, en la Figura 1.10 se muestran dos pasos consecutivos. El método comienza uniendo los dos extremos. Eso nos ocupa una unidad de tiempo. En cambio, lo que nos queda por plegar es la mitad de la longitud inicial. Luego, en el segundo paso, reducimos el problema a la cuarta parte del problema inicial. Está bien claro que este algoritmo es más eficiente que el anterior. En este caso, si el cordel fuera el doble de largo, no tardaríamos el doble para plegarlo, sinó tan sólo una unidad de tiempo más. Este segundo algoritmo es $\Theta(log(n))$. Conviene observar aquí que hay mucho cordel que ni siquiera es tocado por la mano.

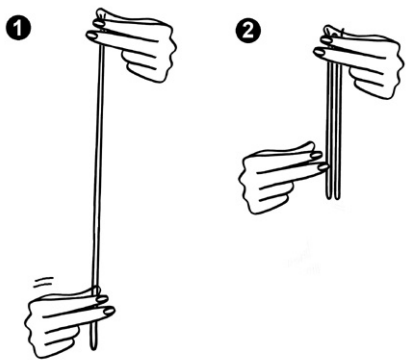


Figura 1.10: *Algoritmo logarítmico para guardar un cordel en la bolsillo.*

Aunque pueda parecer una frivolidad, el hecho de plegar el cordel de una manera o de la otra, para quien haya de plegar diez mil cordeles, puede suponerle un ahorro de tiempo muy importante.

Hemos hablado de los logaritmos. Cambiemos de tema. De las funciones polinómicas podemos decir que el exponente de la potencia nos está revelando el nivel de anidamiento de bucles que tiene el algoritmo. Eso es, un algoritmo $\Theta(n^2)$ es un algoritmo que tiene un bucle dentro de otro bucle. Y $\Theta(n^3)$, un bucle dentro de un bucle dentro de un bucle. Otra manera de verlo es que los algoritmos pertanecientes a $\Theta(n^2)$ tratan cada posible pareja de elementos de la entrada. Los $\Theta(n^3)$, cada posible trio. Y así sucesivamente... Hasta que más que tratar todas las posibles parejas, trios, cuartetos o quintetos, tengamos que tratar ya no los grupos de seis siete u ocho, sinó los grupos de n elementos de entrada. Programas que el número de bucles anidados dentro de bucles dependiera del tamaño de la instancia. Estos tipos de programas ya no son fáciles de implementar. Ultrapasan las cotas polinómicas.

La razón por la cual en la Figura 1.7 la función $n\ log(n)$ está en un color más claro es porque $\Theta(n\ log(n))$ es una eficiencia muy frecuente. En particular, los algoritmos de ordenación de listas acostumbran a pertenecer a $\Theta(n\ log(n))$. Y tanto nos familiarizaremos con ella, que le quitamos los paréntesis y la llamamos "ene log ene", $\Theta(n\ log\ n)$.

El número de subconjuntos que podemos hacer a partir de un conjunto de n elementos es 2^n. Y ¿Por qué aparece un 2 en el número de subconjuntos que se pueden hacer de un conjunto?. Pues porque no se puede hacer un subconjunto sin hacer dos. Hay una curiosidad relativa al creicimiento exponencial 2^n: No hay ninguna hoja de papel en este planeta que se pueda doblar sobre sí misma más de 10 veces. De hecho es normal si lo intentamos imaginar. Si pudiéramos doblar una hoja que midiera un milímetro de grosor (o sea, bastante gruesa, como una cartulina o un cartón) diez veces sobre sí misma, nos quedaría un papel doblado de un metro de grosor, cosa que ya se ve que no puede ser. Probando hacia el otro extremo, un paper de fumar hace unas 30 micras de grosor, o sea, 30 papeles hacen un milímetro. Si probáramos a doblarlo 10 veces, el papel doblado haría 3 centímetros, cosa que también se ve que es imposible. Cierta vez, haciendo este comentario en clase, un alumno quedó con tal curiosidad que la noche siguiente me había enviado un correo electrónico en el que se explicaba que el máximo número de pliegues que se había conseguido (con una lámina de oro, eso sí) era de 12. Efectivamente, cliqué el link y leí el artículo donde se explicaba.

Respeto $n!$, podemos decir que es el crecimiento más grande que nos encontramos en fenómenos de la naturaleza. Crece así el número de ordenaciones que puede tener un lista a medida que crece el número de sus elementos, puesto que para cada nuevo elemento tenemos n posiciones donde colocarlo.

Bien, ya para acabar con la presentación de la notación asintótica, queda por decir que hay que tener en cuenta que no sólo serán éstas las funciones de referencia sinó también cualquier producto de éstas.

Anticipemos en este momento que si de todas las funciones de la Figura 1.7 tuviéramos que hacer dos conjuntos, no cabe duda que la línea divisoria que las

separaría en dos grupos estaría entre las que están por debajo, o igual que las polinómicas, $O(n^k)$, y las que están por encima, o igual, de las exponenciales $\Omega(2^n)$. Si en este momento esta división no queda clara, a lo largo del libro se verá que tiene una importancia decisiva. Y que de hecho, el conocimiento humano del algoritmia tiene una limitación en este sentido. Una frontera.

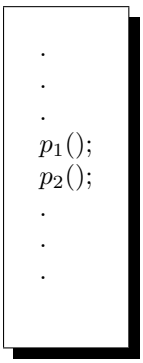
1.5 Análisis de Eficiencia en Algoritmos Iterativos

Una vez definida la herramienta que nos ha de permitir medir los tiempos de ejecución de los algoritmos, se muestra en estas últimas secciones como llevar tanta teoría a la práctica. Ha quedado bien entendido que medir los tiempos de los algoritmos significa clasificar la función de tiempo de respuesta, o tiempo de ejecución en función del tamaño de los datos de entrada, en algún conjunto de los que se han visto en la Sección 1.4, de la página 25. Podemos decir la misma cosa de muchas formas: Medir el tiempo de ejecución, clasificar el tiempo de respuesta, analizar la eficiencia del algoritmo..., en definitiva todo viene a ser lo mismo y se resume diciendo que el algoritmo es $\Theta(f(n))$. En particular se trata de decir cuál es esta $f(n)$, y, cuando sea significativo, distinguir el caso medio del caso peor.

Como es bien sabido, los algoritmos iterativos se estructuran en base a tres tipos de construcciones: La composición secuencial, la sentencia alternativa y la estructura iterativa. Estudiemos pues con detalle cada una de ellas.

1.5.1 Composición secuencial

Clasificamos el tiempo de ejecución de una secuencia de instrucciones considerando tan sólo el máximo de la secuencia. O sea, si tenemos un fragmento de código que sigue la estructura del Algoritmo 1.1, el tiempo correspondiente será $\Theta(max(t_{p_1}, t_{p_2}))$. Este cálculo es sencillo y no debería acarrear ningún problema.

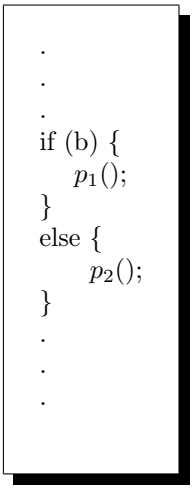


Algoritmo 1.1 *Composición secuencial.*

En este tipo de construciones no habrá distinción entre casos.

1.5.2 Sentencias alternativas

Para las sentencias alternativas no hay mucha cosa nueva que contar. Si tenemos un fragmento como el que se muestra en el Algoritmo 1.2 y nos piden su tiempo de ejecución, la respuesta será, como en el caso anterior, $\Theta(max(t_{p_1}, t_{p_2}))$. En el Algoritmo 1.2, la variable b significa una condición booleana.



Algoritmo 1.2 *Sentencia alternativa.*

Con las sentencias alternativas es probable que valga la pena distinguir entre caso mejor o caso medio y caso peor, cuando la diferencia entre t_{p_1} y t_{p_2} sea significativa, aunque en general se calculará la eficiencia tal y como se ha dicho.

Siempre que en un algoritmo se requiere distinguir entre caso medio y caso peor cuando hay que calcular su eficiencia es debido a que el flujo de ejecución viene gobernado por los datos y no solamente por su tamaño. En esas situaciones, será necesario entrar en la distinción de casos cuando la expresión lógica b involucre valores representativos del contenido de la entrada en sentido cualitativo más que cuantitativo. Les sentencias alternativas son la vía más clara de introducir estas imprecisiones en el cálculo de las eficiencias.

De hecho, este tipo de sentencias introducen comportamientos irregulares en función de las condiciones y, de las tres estructuras que se están estudiando para los algoritmos iterativos, son las que menos elegancia otorgan a los programas que las contienen ya que son exactamente lo contrario de abstracción. Directamente, introducir sentencias alternativas en los programas cuando no son estrictamente necesarias provoca un detrimento importante de la calidad del código que se está produciendo.

1.5.3 Estructuras iterativas

La inmensa mayoría de programadores noveles no entienen por qué se dispone de dos estructuras iterativas, la *for* y la *while* y muchos otros que llevan años programando tampoco. Pues bien, aquí se exponen dos razones.

En primer lugar, aclaremos que un *for* siempre puede ser implementado con un *while* y un contador. En cambio, un *while* no siempre puede ser implementado con un *for*. Sólo por eso, ya hace sospechar que iguales del todo no lo son, y debería indicar que siempre que se pueda se utilizará un *for*.

La pregunta, llegados a este punto es: ¿Por qué existe el *for*?. De las dos razones que seguidamente se dan, la primera no tiene nada que ver con la eficiencia.

- Razón práctica: Cuando los programas crecen con el tiempo sufriendo un mantenimiento no siempre fácil de gestionar, si hay *while*s donde podría haber *for*s, debido a la independencia real que hay entre el contador y el código de la estructura iterativa, el incremento del contador puede quedar situado en diferentes partes del interior del bucle. Esto provoca un descontrol que en ningún caso resulta conveniente. Si en las diferentes actualizaciones del código el cuerpo del bucle crece mucho, introduciéndose sentencias alternativas en medio, el incremento puede llegar a quedar colocado en lugares erróneos provocando fácilmente bucles infinitos.

  Hasta antes del uso de la memoria dinámica, trabajando con lenguajes como fortran o cobol, en los programas informáticos, el problema más grave era que los programas se colgaban. Este problema se convierte en altamente incómodo cuando los procesos tardan mucho rato debido a la complejidad del problema. Entonces, no había nada peor que llevar

un tiempo grande esperando los resultados, y no tener la certeza de si quizás el programa se había colgado. Más tarde, cuando se empezó a trabajar en lenguaje C, un nuevo problema consiguió imponerse como el peor. Los punteros descontrolados, mal reservados o mal liberados. Ese es el problema más grave que pueden tener los programas actualmente.

Definitivamente, si un programa no tiene ningún *while*, es imposible que se cuelgue. Y cuando en lugar de un solo programa hablamos de una batería de procesos que hay que ejecutar en serie, entonces si el proceso global se cuelga, podemos descartar que sea en alguno de los programas libres de *while*s, y sólo será necesario revisar los que efectivamente sí que incorporan *while*s.

La segunda razón, en cambio, está intimamente ligada a la eficiencia.

- Razón filosófica: Los bucles *for* involucran el tamaño de los datos de entrada n, ligado a la definición del cálculo de la eficiencia. En otras palabras, los *for*s tienen que ver con la cantidad de datos de la entrada. Los *while*s, en cambio, tienen que ver con el contenido de los datos, con lo que cualitativamente son.

En el Algoritmo 1.3 se muestra un esquema clásico de cualquier procedimiento iterativo. Se trata de un bucle *for*, que es el más sencillo de los bucles que vemos. En la próxima sección se verá en detalle el cálculo de la eficiencia, implementando un bucle *for* en un *while*.

Del análisis de la eficiencia del Algoritmo 1.3 se desprende que el tiempo de ejecución pertenece a $\Theta(n)$, siempre que el tiempo de cada uno de los procedimientos p_i sea constante, $T_{p_i} = \Theta(1)$.

```
.
.
.
for (int i=0; i<n; i++){
      p_i();
}
.
.
.
```

Algoritmo 1.3 *Estructura iterativa.*

Si no es así, o sea, que $T_{p_i} = \Theta(f)$ para alguna $f = f(i)$, entonces la cosa se complica. En tal caso se puede acotar con no tanta precisión utilizando la notación $O()$.

$$T(n) = \sum_{i=0}^{n-1} T_{p_i} = O(n \max_{0 \le i < n} \{T_{p_i}\}).$$

Algoritmo iterativo en detalle

Contemos el número de operaciones elementales que hace el fragmento de código del Algoritmo 1.4. Para eso, llamamos c al tiempo de realizar una. Y consideramos t el tiempo que tarda la operacion $p(i)$, independientemente de i.

Observad que suponer que $t = p(i)\ \forall i$ es una suposición fuerte.

En el margen derecho del Algoritmo 1.4 se muestra una columna con los números de operaciones elementales que realiza cada línea del código. Se supone que las operaciones elementales, suma, resta (o sea comparación), multiplicación, asignación, llamada, retorno, salto,... tardan un tiempo constante que, como se ha dicho, llamamos c.

Como se puede ver, la asignación inicial consta de una sola operación elemental. La segunda línea es la cabecera del bucle. En ella hay una comparación que se realiza $m+1$ veces, de las cuales las m primeras el resultado será *cierto* y la última, *falso*. En el interior del bucle tenemos una llamada a una instrucción externa que se realiza m veces, igual que el incremento de la línea siguiente y el salto en el flujo de ejecución que supone el cierre del bucle.

```
.
.
.
int i=1;                 c
while (i≤m) {            (m + 1) c
        p(i);            mt
        i++;             mc
}                        mc
.
.
.
```

Algoritmo 1.4 *Cálculo en detalle del tiempo para una estructura iterativa.*

La suma de todas las operaciones elementales da $(t+3c)m+2c$. Razonablemente podemos suponer que t es mucho mayor que c. Entonces, en conjunto,

este fragmento de código será $\Theta(mt)$, siempre que $m = m(n)$. Es decir, siempre que el límite del bucle, m, dependa del tamaño de la instancia... A ver, a no ser que estemos hablando de un bucle para escribir las tablas de multiplicar, en general, que $m = m(n)$ sucederá siempre. Es decir, el número de veces que hay que ejecutar un bucle acostumbra a depender del tamaño de los datos de la instancia del problema que se está resolviendo.

1.5.4 Eficiencia de la ordenación por selección

Como ejemplo, nos disponemos a analizar la eficiencia de uno de los algoritmos más conocidos de ordenación. La ordenación por selección no es conocida precisamente por su eficiencia, que deja mucho que desear, sinó por su simplicidad.

Para ordenaciones crecientes, a cada paso se seleciona el mínimo de los no ordenados, y se coloca en el siguiente lugar de la parte ordenada. Es poco eficiente porque la búsqueda de este mínimo se hace de manera secuencial. Eso supone desaprovechar mucho trabajo. Es decir, cuando un número es más pequeño que otro, y este otro es más pequeño que un tercer número, no sacamos ningún provecho de la segunda comparación entre el primero y el tercero. O sea, si x es menor que y, e y es menor que z, comparar x con z no sirve de nada, y es trabajo perdido.

En el Algoritmo 1.5 se puede ver el código de este algoritmo.

```
void ordenacion_por_selecion(int n, int T[ ])
{
    for (int i=0; i<n-1; i++) {
        int minj = i;
        int minx = T[i];
        for (int j=i+1; j<n; j++) {
            if (T[j]<minx) {
                minj = j;
                minx = T[j];
            }
        }
        T[minj] = T[i];
        T[i] = minx;
    }
}
```

Algoritmo 1.5 *Ordenación por selección.*

Para clasificar el tiempo de ejecución del Algoritmo 1.5, llamemos b al tiempo necesario para hacer las dos inicializaciones, y las dos asignaciones finales del

bucle más externo, el del índice i. Al coste temporal de realizar la sentencia alternativa del bucle más interno, que también es constante, lo denotamos por c.

Entonces,

$$\begin{aligned} T(n) &= \sum_{i=0}^{n-2} b + (n-i)c \\ &= \sum_{i=0}^{n-2} (b+cn) - c\sum_{i=0}^{n-2} i \end{aligned}$$

El primer sumatorio no depende de i, es sencillamente $(n-1)$ veces $(b+cn)$.

El segundo sumatorio es la suma de los términos de una progresión aritmética. El primer término es el 0, el último el $(n-2)$. Hay $(n-1)$ términos. Con todo, pues,

$$\begin{aligned} T(n) &= (n-1)(b+cn) - c(0+n-2)(n-1)/2 \\ &= cn^2/2 + (2b-c)n/2 - b = \Theta(n^2) \end{aligned}$$

Del cálculo de la eficiencia anterior conviene familiarizarse con la suma de los $(n+1)$ primeros términos de la sucesión aritmética $a_i = i$, para $i = 0, \ldots, n$, ya vista en la Sección 1.1.2 de la página 8. El primero más el último, multiplicado por la mitad de los que hay. En concreto, diremos números triangulares a los que resulten de estas sumas.

$$\sum_{i=0}^{n} i = (0+n)(n+1)/2 = \sum_{i=1}^{n} i = (1+n)n/2$$

De hecho, que la eficiencia de la ordenación por selección sea $\Theta(n^2)$ ya se puede suponer a partir del conocimiento de que hacemos comparaciones entre cualquier pareja de los n números de la entrada. Y tal como se ha dicho en la Sección 1.4, al final del apartado *Funciones de referencia habituales* de la página 34, los algoritmos $\Theta(n^2)$ consideran cada posible pareja de elementos de la entrada.

1.5.5 Eficiencia de la ordenación por inserción

Como segundo ejemplo y para cerrar esta sección de algoritmos iterativos analizaremos la eficiencia de la ordenación por inserción. Además, en este caso se podrá distingir entre casos medio y peor, aunque en definitiva, resultará ser $\Theta(n^2)$ en los casos interesantes. La identificación del caso peor añade interés a esta sección.

Así como el algoritmo de ordenación por selección visto en la Sección 1.5.4 trabaja selecionando el mínimo siguiente a ordenar de entre todos los elementos

no ordenados, la ordenación por inserción soluciona el problema de ordenar de manera inversa: Considera que el primer número ya está ordenado, y entonces ordena el siguiente, o sea el segundo en el primer caso, respeto la parte ordenada, que en el primer caso será tan sólo el primer número. En la Sección 3.4.1, de la ordenación por fusión, se habla con más detalle de lo que son los problemas inversos.

```
void ordenacion_por_insercion(int n, int T[ ])
    for (int i=1; i<n; i++) {
        int x = T[i];
        int j = i - 1;
        while (j>=0 && x<T[j]) {
            T[j+1] = T[j];
            j = j - 1;
        }
        T[j+1] = x;
    }
}
```

Algoritmo 1.6 *Ordenación por inserción.*

Está bien claro que el *while* del Algoritmo 1.6 no puede ser sustituido por un *for*. Este es un ejemplo de lo que se ha dicho en la Sección 1.5.2 respeto la relación entre la necesidad de distinguir casos mejor, medio y peor, y el hecho de que exista control del flujo gobernado por el contenido de los datos de entrada. Cuando ocurre esto, se requiere hacer suposiciones sobre este contenido y esto provoca la necesidad de distinción de casos. Por todo esto, para hacer el análisis de eficiencia del Algoritmo 1.6 comenzamos suponiendo...

caso mejor: Aunque tenga poca trascendencia, contemplamos el caso mejor. Este caso se dará cuando la segunda parte de la condición del *while*, o sea $x < T[j]$, sea siempre falsa (fijaos bien que para distinguir casos mejor y peor siempre hacemos suposiciones sobre los valores de los datos). Es decir, cuando x siempre sea mayor que $T[j]$. Teniendo en cuenta que $x = T[i]$ y siempre j es menor que i, estamos diciendo que el caso mejor se dará cuando $T[i]$ sea mayor que $T[j]$, $\forall i, j$ tales que $i > j$, $i, j \in \{0, .., n-1\}$. Total, cuando el vector ya esté ordenado antes de empezar. En este caso, el cuerpo del *while* no se ejecutará nunca provocando que el bucle interior completo tarde $\Theta(1)$. Entonces, la ordenación completa tardará $\Theta(n)$. Así pues, el análisis de eficiencia para este caso debería concluir diciendo que el caso mejor es aquél en el que el vector de entrada ya está ordenado, que el algoritmo tarda $\Theta(n)$.

caso peor: En el caso peor, siempre saldremos del *while* porque la j llegará a 0. Es el caso completamente inverso del anterior. Así pues, el caso peor es

cuando el vector inicial está ordenado al revés, o sea, decrecientemente en el Algoritmo 1.6. En el caso de que el contenido del vector de entrada esté ordenado decrecientemente, el número de operaciones elementales será la suma de los términos de una progresión aritmética, o lo que es lo mismo, el primero más el último multiplicado por la mitad de los que haya. O sea,

$$T(n) = \sum_{i=1}^{n-1} i = (1 + n - 1)(n - 1)/2 = n(n - 1)/2 = \Theta(n^2)$$

ya que dentro del *for*, el *while* se ejecutará i veces en cada iteración.

caso medio: Para el caso medio hay que suponer una distribución de probabilidad, cosa que tendrá un fuerte impacto en el análisis. Supongamos una distribución uniforme.

Llamemos k a la variable aleatoria que nos dice en qué posición final va el actual elemento a ordenar. Esta posición ha de estar dentro del intervalo $[1, i]$ en cada iteración, o sea, dentro de la parte ordenada. Es decir, supongamos que $k \sim U[1, i]$, de manera que la probabilidad que el elemento actual $T[i]$ vaya a una posición concreta k_0, $1 \leq k_0 \leq i$, sea $P(k = k_0) = 1/i$. Entonces, el número de operaciones elementales en cada iteración, que denotamos por c_i será

$$c_i = 1/i \sum_{k=1}^{i} k = (i + 1)/2,$$

es decir, la probabilidad de tener que hacer k comparaciones multiplicado por k. Y, en total,

$$T(n) = \sum_{i=1}^{n-1} c_i = (n - 1)(n - 4)/4 = \Theta(n^2)$$

A pesar de que la ordenación por inserción tiene una eficiencia del mismo orden que la ordenación por selección, es mejor en el caso mejor, y ligeramente más eficiente en el caso medio, aunque esta mejora de eficiencia queda absorvida en la constante oculta de la notación asintótica.

1.6 Análisis de Eficiencia en Algoritmos Recursivos

Una función recursiva es aquella que en algún caso se llama a sí misma. Se podría imaginar una función recursiva que no distinguiera casos. Debería ser una función que se ejecutara eternamente. Se podría tratar, por ejemplo, de

imprimir todos los números naturales. Eso significa imprimir tantos como sea posible.

Para este objetivo se podría implementar una función recursiva como la del Algoritmo 1.7. En la realidad, si en un programa hubiera una llamada a *cuenta(0)*, se provocaría un error en el sistema operativo de desbordamiento de la pila por excesivo anidamiento de llamadas. Si no fuera así, entonces sería un error por desbordamiento de datos en el momento en que n superase el valor 2^{32} con los lenguages de programación actuales, que normalmente alamcenan una variable entera utilizando 32 bits.

```
void cuenta(int n)
{
    imprimir(n);
    cuenta(n+1);
}
```

Algoritmo 1.7 *Función recursiva infinita.*

Esta reflexión pues, tan sólo es útil para ilustrar la necesidad de distinguir casos en las funciones recursivas. Llamamos caso trivial aquél en que la función no se llama, y la recursividad se acaba.

La expresión analítica que describe una función recursiva es una recurrencia.

Definición 1.8 Recurrencia. *Se llama recurrencia a una expresión formal que relaciona una función real de r variables enteras, $f : \mathbb{N}^r \to \mathbb{R}$, con la misma función para otros valores de las variables. Debe ser definida por intervalos o casos de las variables, y debe existir alguno de los casos (caso trivial) donde la expresión recurrente no aparezca.*

Por ejemplo, con $r = 1$,

$$f(n) = \begin{cases} k & si\ n{=}0 \\ 2f(n-1) & si\ n{>}0 \end{cases}$$

Así pues, si tal como se ha dicho, la función recursiva debe tener casos, entonces necesariamente debe tener unos datos de entrada de los cuales dependan esos casos. Por tanto, toda función recursiva debe tener parámetros. La sucesión de valores que toma el parámetro de la función recursiva debe converger necesariamente a alguno de los valores que condiciona el flujo de la ejecución hacia algún caso trivial. Habitualmente estas sucesiones serán decrecientes, y frecuentemente el caso trivial será cuando el parámetro valga cero o uno.

La manera cómo decrece el parámetro sobre el que se soporta la recursividad nos define dos tipos de recursividad que llamamos substractora, y divisora.

1.6.1 Recursividad Substractora: Teorema Maestro I

Un algoritmo recursivo utiliza recursividad sustractora cuando en términos generales la función f que implementa se puede describir como

$$f(n) = af(n-c) + g(n) \tag{1.4}$$

para algunos $a, c \in \mathbb{N}$ y alguna función $g \in \mathbb{L}$.

En la expresión (1.4), a parte de la variable n, hay otros parámetros que caracterizan la recursividad. Sus sentidos son los siguientes,

- a es el número de llamadas recursivas.
- f es la función recursiva propiamente dicha.
- c es el decremento del parámetro de recursividad entre llamadas sucesivas.
- g significa el conjunto de las otras cosas que hace el algoritmo, que no sean llamadas recursivas.

Se trata de una recursividad en la que el decremento al que se somete el parámetro entre llamada y llamada es constante.

Para analizar la eficiencia de este tipo de funciones, ya se puede suponer que el tiempo seguirá una recurrencia parecida, $T(n) = aT(n-c) + T_g(n)$, donde $T_g(n)$ es el tiempo asociado a las operaciones que se realicen en $g(n)$.

Analizar la eficiencia significa pues resolver una ecuación en diferencias. No es sencillo, y no se le supone al lector el conocimiento del análisis matemático que hay detrás. De hecho, este tipo de ecuaciones vienen a ser la versión discreta de lo que en el análisis continuo son las ecuaciones diferenciales.

No obstante, para no quedarse de brazos cruzados, se ilustra seguidamente un método poco riguroso pero facilmente comprensible. Es conocido como método de incrustación, ya que se trata de sustituir el término $T(n-c)$ de la derecha de la ecuación recurrente por la definición de la misma ecuación, considerando $T_g(n) = \Theta(1)$. Estas sustituciones se hacen sucesivamente hasta llegar a consi-

derar el caso trivial, $T(0)$. Es decir,

$$\begin{aligned}
T(n) &= T(n-c) + \Theta(1), \text{ pero como } T(n-c) = T((n-c)-c) + \Theta(1), \text{ entonces} \\
&= T((n-c)-c) + \Theta(1) + \Theta(1) = T(n-2c) + 2\Theta(1) \\
&= T(n-3c) + 3\Theta(1), \\
&\dots \\
&(n/c \text{ veces}) \\
&\dots \\
&= T(0) + (n/c)\Theta(1) = \Theta((n/c))
\end{aligned}$$

O sea, después de incrustar repetidamente la definición de $T(n)$ dentro de sí misma n/c veces, se llegará a una expresión como $T(n) = T(0) + \dots$. A partir de ahí, y siempre para el caso que $T_g(n)$ sea $\Theta(1)$, tenemos $T(n) = \Theta(n)$.

De todas formas, a pesar de no tener bastante conocimiento matemático para resolver ecuaciones en diferencias, la recurrencia que describe las recursividades substractoras es siempre la misma, y por tanto podemos utilizar siempre el Teorema 1.1. No debemos confundir las funciones f y g de la recurrencia analizada, con las funciones f y g presentes en el Teorema 1.1.

El tiempo de un algoritmo de estructura recursiva substractora con expresión genérica

$$T(n) = \begin{cases} f(n) & si\ n \le n_0 \\ aT(n-c) + g(n) & si\ n > n_0 \end{cases}$$

con $f, g \in \Theta(n^k)$, para alguna $k \in \mathbb{N}$, puede ser clasificado directamente con

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

Teorema 1.1 *Teorema Maestro I.*

Los actores del Teorema 1.1, significan:

n_0: Máximo valor del parámetro n que condiciona el flujo hacia un caso trivial.

$f(n)$: Tiempo que tarda el algoritmo en los casos triviales.

a: Número de llamadas recursivas (en ejecución!).

c: Decremento que modifica el parámetro entre dos llamadas sucesivas.

$g(n)$: Tiempo que tarda el algoritmo en los casos recursivos excepto las llamadas recursivas propiamente dichas.

k: Grado del polinomio con el que acotamos el tiempo requerido por las funciones f y g.

Cuando no aparezca el tamaño de la instancia, n, en el código que analizamos, deberemos empezar definiéndola a partir de los parámetros de entrada. Es decir, para proceder a la identificación de los parámetros del Teorema 1.1 se debe haber definido previamente qué significa el tamaño de la instancia.

Fijaos que se supone que tanto $f(n)$ como $g(n)$ son $\Theta(n^k)$ para alguna k. Esta suposición es consecuencia de la regla de la suma de la notación asintótica explicada en la Sección 1.4, apartado de *Propiedades de los conjuntos de funciones*, de la página 28. Eso significa que si f o g tardasen tanto que no se pudieran acotar por una potencia de n (y por tanto como mínimo se podrían acotar por $O(2^n)$) entonces la parte recursiva no tendría importancia alguna en la eficiencia, ya que toda ella vendría dominada por la parte no recursiva.

Observad también que el primer caso de las soluciones para $T(n)$ no tiene demasiada utilidad, ya que $a \in \mathbb{N}$, por definición. Y por tanto, $a < 1$ significa que no hay llamadas recursivas. De totas formas, analizadlo. Si no hay llamadas recursivas, el algoritmo tarda como f o como g, el máximo de los dos.

Por otra parte, en los dos primeros casos de estas soluciones no interviene el parámetro c. Esto es fruto de la definición de límite en la definición de la notación asintótica. Diferentes c's nos darían diferentes constantes ocultas. Hay un paralelismo claro entre una llamada recursiva substractora y un bucle *for*. Ambos incrementan el exponente del polinomio argumento de la notación. En un *for* no importa el incremento ya que queda absorvido por la constante oculta. En una función recursiva substractora exactamente lo mismo.

Finalmente, observad que el caso $a > 1$ nos conduce a eficiencias terribles. Hay que evitar por todos los medios este tipo de algoritmos.

Una de las cosas que han de quedar más claras después de la lectura de este libro, y que es una cuestión latente en todos los capítulos, es que no podemos considerar completamente resuelto un problema si el tiempo que tarda para una instancia de tamaño n no se puede acotar polinómicamente con n. Y es que en el fondo, proponerse hacer una cosa para mañana, y no hacerla, son hechos bastante parecidos.

1.6.2 Eficiencia del cálculo del factorial

Es bien sabido que el factorial de un número natural es el productorio de todos los números naturales inferiores o igual a él. Éste es el número de ordenaciones

diferentes que puede tener una lista, o el número de maneras de hacer n recados cuando se sale de casa, que es lo mismo. Permutaciones, en definitiva.

$$factorial(n) = \prod_{i=1}^{n} i$$

Tiene su lógica. Cuando pretendemos ordenar n elementos, tenemos n posibilidades diferentes para escoger el primero. Y entonces, para cada una de estas posibilidades, tenemos $n-1$ posibilidades para el segundo. O sea, que tenemos, de momento, $n(n-1)$ posibilidades de segundos elementos. Para el tercero, serán $n-2$, y así sucesivamente.

En el Algoritmo 1.8 se puede observar una función recursiva para realizar el cálculo del factorial de un número.

```
int factorial(int n)
{
    if (n≤ 1) return 1;
    return n * factorial(n-1);
}
```

Algoritmo 1.8 *Cálculo del factorial.*

Como el tamaño n ya está presente en el código, para analizar su eficiencia pasamos directamente a describir la recurrencia para $T(n)$. Esto es,

$$T(n) = \begin{cases} \Theta(1) & si\ n \leq 1 \\ T(n-1) + \Theta(1) & si\ n > 1 \end{cases}$$

Y a partir de ahí, tan sólo identificando los tres parámetros a, c, y k para saber en cuál de los tres casos del Teorema Maestro I nos encontramos, ya tendremos el análisis realizado. No es difícil ver que $a = 1$, ya que tan sólo se produce una llamada recursiva; $c = 1$, ya que el decremento del parámetro de la llamada recursiva es 1; y, si contamos las operaciones elementales en los flujos del caso trivial, o del caso recursivo excepto la llamada, tenemos, para el caso trivial dos (la comparación y el *return*), y para el flujo asociado a la llamada recursiva, también dos (el producto y el *return*). Entonces, $\Theta(n^k) = 2$ significa que $\Theta(n^k) = \Theta(1)$. Luego, $k = 0$.

Por fin, como estamos en el caso central del Teorema 1.1, ya que $a = 1$, podemos concluir directamente que la función $factorial(n) = \Theta(n)$, cosa que no nos ha de sorprender si pensamos en la versión iterativa del algoritmo para calcular el factorial.

El hecho de implementar un algoritmo en distintas versiones, recursiva o iterativa, en ningún caso alterará la eficiencia.

1.6.3 Eficiencia del cálculo de los números de Fibonacci


Leonardo da Pisa

Leonardo de Pisa (1170-1250) fue un matemático nacido en Pisa que divulgó por Europa el sistema de numeración decimal basado en la notación posicional y el uso de 10 cifras, así como el uso de un elemento neutro o número cero.

Por otra parte definió lo que se ha conocido como la sucesión de Fibonacci basado en un estudio de la reproducción de conejos. Esta sucesión es facilmente deducible teniendo en cuenta que una pareja de conejos tarda un mes desde que nace hasta ser una pareja madura con capacidad de reproducción. Un vez madura, tarda un mes más para poder reproducirse pariendo una nueva pareja. Si el proceso comienza con una pareja y se cuenta el número de parejas que hay cada mes, los números que se obtienen son:

1. El primer mes hay una pareja de conejos sin capacidad de reproducción, es decir, jovencita. (..)
2. El segundo mes sigue habiendo una sola pareja, que ya ha madurado y por tanto puede reproducir. (oo)
3. El tercer mes hay dos parejas, puesto que la primera ha criado una pareja de conejos jovencitos. (oo,..)
4. El cuarto mes hay tres parejas, ya que la primera madura ha vuelto a parir otra pareja, y la otra ha tenido tiempo de crecer. (oo,oo,..)
5. El quinto mes ya son cinco parejas, ya que después del cuarto mes, la joven inicial ya ha parido y la pareja más vieja ha parido la tercera pareja, mientras la segunda crecía.(oo,oo,oo,..,..)

Y así, ir haciendo. Los primeros términos de la sucesión son pues 1, 1, 2, 3, 5, 8, 13,... de manera que los dos primeros números son 1, y a partir del tercero son la suma de los dos anteriores. Así pues, la sucesión de Fibonacci tiene una definición que por naturaleza parece recursiva.

$$fibonaci(n) = \begin{cases} 1 & si\ n \leq 2 \\ fibonaci(n-1) + fibonaci(n-2) & si\ n > 2 \end{cases}$$

A pesar de tener una definición tan naturalmente recursiva, como todos los algoritmos también se puede implementar iterativamente. En este punto, aflora algún misterio en el que no adentraremos. El término general para definir la misma sucesión iterativamente involucra la sección áurea. El enésimo término de Fibonacci descrito sin necesidad de recurrencias resulta bastante complicado.

$$fibonaci(n) = (1/\sqrt{5})(\phi^n - (-\phi)^{-n}),$$

siendo phi, ϕ, la sección áurea $\phi = (1 + \sqrt{5})/2 = 1.6180...$, número cargado de simbolismos utilizado profusamente en diversas disciplinas. Además, la sucesión de Fibonacci satisface otras interesantes propiedades como, por ejemplo, que la suma de los $n - 1$ términos más 1 es igual al término $n + 1$, para tota n.

$$fibonaci(n + 1) = 1 + \sum_{i=1}^{n-1} fibonaci(i).$$

Bien, volviendo al tema, en el Algoritmo 1.9 se puede observar una función recursiva para realizar el cálculo del término enésimo de la sucesión de Fibonacci.

```
int fibonacci(int n)
{
    if (n<=2) return 1;
    return fibonaci(n-1) + fibonaci(n-2);
}
```

Algoritmo 1.9 *Cálculo de los números de Fibonacci.*

Para analizar su eficiencia es necesario, como antes, comenzar describiendo la recurrencia para $T(n)$, puesto que el tamaño n ya se explicita en el código. Eso es,

$$T(n) = \begin{cases} \Theta(1) & si\ n \leq 2 \\ T(n-1) + T(n-2) + \Theta(1) & si\ n > 2 \end{cases}$$

Y a partir de aquí, tan sólo identificando los tres parámetros a, c, y k para saber en cuál de los casos del Teorema Maestro I nos encontramos, ya tendremos el análisis realizado. En el Algoritmo 1.9, $a = 2$, ya que hay dos llamadas recursivas. Sin embargo, con el parámetro c tenemos un problema, ya que el Teorema 1.1 sólo contempla la existencia de un parámetro c. Lo que podemos hacer es dar una cota superior al tiempo. Supongamos que en lugar del término $T(n-2)$, tuviéramos un segunda vez $T(n-1)$, que aumentaría el tiempo global. Así conseguiremos al menos una cota superior. Convenimos pues que $c = 1$. Luego, tenemos dos operaciones elementales en los flujos no recursivos. Y si $\Theta(n^k) = 2$, entonces $\Theta(n^k) = \Theta(1)$ y por tanto, $k = 0$.

Concluimos el análisis utilizando el tercer caso del Teorema Maestro I, que supone que $a > 1$. Clasificamos la eficiencia del Algoritmo 1.9 notando que $T(n) = O(2^n)$. Desgraciadamente, a partir de esta eficiencia debemos admitir que este algoritmo para resolver el término enésimo de Fibonnacci es inadmisible a pesar de su sencillez. En efecto, estamos frente un caso del algorísmia en el que la sencillez va reñida con la eficiencia. Cosa extraña. En capítulos posteriores veremos otros algoritmos más eficientes para hacer este mismo cálculo.

1.6.4 Recursividad Divisora: Teorema Maestro II

Se dice que un algoritmo recursivo utiliza recursividad divisora cuando en términos generales la función que implementa se puede describir como

$$f(n) = af(n/b) + g(n) \tag{1.5}$$

para algunos $a, b \in \mathbb{N}$ y alguna función $g = g(n)$.

En la expresión (1.5), a parte de la variable n, hay tres parámetros que caracterizan la recursividad.

- a es el número de llamadas recursivas.
- f es la función recursiva propiamente dicha.
- b es el factor de reducción del parámetro de recursividad entre llamadas sucesivas.
- g significa el conjunto de las otras cosas que hace el algoritmo que no sean las llamadas recursivas.

A partir de la recurrencia propia de las recursividades divisoras (1.5) puede suponerse que en general resultan más prometedoras que las anteriores.

Ya se ve que la posibilidad de desarrollar algoritmos más eficientes con este segundo tipo de recursividad es más fácil, ya que aquí la secuencia de valores que toma el parámetro sobre el qué se fundamenta la recursividad es de naturaleza geométrica, y por tanto convergerá más rapidamente a los casos triviales que cuando lo hacía aritméticamente en el caso de las recursividades substractoras.

Como antes también, el tiempo requerido por un algoritmo con esta estructura sigue una recurrencia parecida, $T(n) = aT(n/b) + T_g(n)$.

Nos encontramos de nuevo en la necesidad de resolver una ecuación en diferencias. La resolución por el método de la incrustación es de forma parecida al caso de la recursividad substractora, aunque el número de llamadas necesarias para llegar al caso trivial ya no es n/c sinó $log_b(n)$. Este análisis genérico, por el método de la incrustación, de la eficiencia de los algoritmos que utilizan recursividad divisora, se expone a continuación. En este cálculo se supone que $T_g(n) = \Theta(1)$.

$T(n) = T(n/b) + \Theta(1)$, pero como $T(n/b) = T((n/b)/b) + \Theta(1)$, entonces
$T(n) = T((n/b)/b) + \Theta(1) + \Theta(1) = T(n/b^2) + 2\Theta(1)$
$= T(n/b^3) + 3\Theta(1)$,
...
($log_b(n)$ veces)
...
$= T(1) + log_b(n)\Theta(1) = \Theta(log_b(n))$

Este análisis, como antes, es meramente ilustrativo. No tiene rigor alguno, y sólo representa una visión en la que conviene reflexionar. De hecho, si este razonamiento resulta ser un apéndice, es gracias a que a lo largo del tiempo ha habido quién se ha dedicado a las ecuaciones en diferencias finitas con suficiente profundidad como para establecer un puente en el discurso que aquí se expone, [1] o [16]. Puente sobre el que pasamos con mucho gusto.

En definitiva, para resolver este segundo tipo de recurrencias utilizaremos el Teorema Maestro II, asociado a la recursividad divisora.

El tiempo de un algoritmo de estructura recursiva divisora con expresión genérica

$$T(n) = \begin{cases} f(n) & si\ n \leq n_0 \\ aT(n/b) + g(n) & si\ n > n_0 \end{cases}$$

con $f, g \in \Theta(n^k)$ para alguna $k \in \mathbb{N}$, puede ser clasificado directamente con

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

Teorema 1.2 *Teorema Maestro II.*

Como ya se ha dicho, probablemente en las funciones que analizemos puede no aparecer ninguna variable que se llame n. Si es así, entonces no queda suficientemente explícito cual es el tamaño de los datos sobre el que soportamos la definición de eficiencia. Deberemos comenzar el análisis diciendo claramente qué es n. Una vez establecida la definición del tamaño de la instancia, procederemos a identificar los parámetros del teorema.

Los actores del Teorema 1.2 son

n_0: Máximo valor del parámetro n que condiciona el flujo hacia un caso trivial.

$f(n)$: Tiempo que tarda el algoritmo en los casos triviales.

a: Número de llamadas recursivas (en ejecución!).

b: Factor de reducción del tamaño del problema entre dos llamadas sucesivas.

$g(n)$: Tiempo que tarda el algoritmo en los casos recursivos excepto las llamadas recursivas propiamente dichas.

k: Grado del polinomio con el que acotamos el tiempo requerido por las funciones f y g.

Para la presentación del Teorema Maestro II hay en la literatura una versión en la qué se define $\alpha = log_b(a)$. Entonces, la casuística de entrada se pone en función de esta α. Es decir, cuando en el Teorema 1.2 mostrado aquí se dice "$si\ a < b^k$", con este nuevo parámetro se dice "$si\ \alpha < k$". Claramente las dos maneras son equivalentes. Aquí se evita el uso del parámetro α porque se considera que no es estrictamente necesario y así, se simplifica la comprensión.

Atención de nuevo en que se supone que toda la parte no recursiva del algoritmo es polinómica. Es decir, el tiempo requerido por esas partes se puede acotar por una potencia del tamaño de los datos, $\Theta(n^k)$. Como ya se ha dicho para el caso de la recursividad substractora, si no fuera así, o sea, si las partes no recursivas tardaran $\Theta(2^n)$ o más aún, entonces por la regla de la suma, la parte recursiva no trascendiría en la eficiencia global que vendría dominada por estas otras partes.

Observad también que con las recurrencias divisoras en ningún caso toparemos con eficiencias tan malas como en las substractoras. No hay ningún caso del Teorema 1.2 que nos conduzca a eficiencias exponenciales.

1.6.5 Eficiencia de la búsqueda dicotómica

Dicotomía quiere decir bifurcación en dos. Este procedimiento de búsqueda es conocido también como búsqueda binaria, búsqueda booleana o búsqueda lógica. Sin duda, éste es el algoritmo con la mejor eficiencia, $\Theta(log(n))$, que nos encontraremos en toda la teoría del algoritmia. La razón de esto, de tener una eficiencia inferior a $\Theta(n)$ se puede explicar teniendo en cuenta que la búsqueda dicotómica ni siquiera trata cada elemento de la entrada. Ni los lee ni los escribe, ni los considera para nada. Así puede reducir el problema a la mitad de su tamaño en cada paso, ignorando la mitad que no interesa.

Para dejar bien claro el funcionamento del método de la búsqueda dicotómica imaginemos el siguiente diálogo entre dos personas:

- Piénsate un número entre el 1 y el 10!
- Ya lo tengo (...piensa el 6)
- ¿Es mayor que 5?
- Sí!
- ¿Es mayor que 7 y medio?
- No!
- ¿Es mayor que 6 coma 25?
- No!
- El 6!

Observad que el número máximo es conocido de entrada (dice piénsate un número entre 1 y 10). Fijaos también en que el segundo personaje, el que responde, en todo momento dice si el número que ha pensado es mayor o no que el que le pregunta. O sea, éste procedimiento requiere que el espacio de búsqueda esté ordenado. Es más, sólo ordenamos las cosas para poder realizar así las búsquedas. Y finalmente, daos cuenta de que el número de preguntas necesarias es el logaritmo en base 2, redondeado al alza, del número máximo inicial. Es decir, la búsqueda dicotómica se puede utilizar sobre un conjunto de elementos de los cuales necesariamente hay que saber cuántos hay. Además, también se requiere que los elementos estén ordenados.

```
int busqueda_dicotomica(double T[ ], int i, int d, double x)
{
    if (i<d) {
        int m = (i+d)/2;
        if (x>T[m]) return busqueda_dicotomica(T,m+1,d,x);
        if (x<T[m]) return busqueda_dicotomica(T,i,m-1,x);
    }
    if (x==T[i]) return i;
    else return -1;
}
```

Algoritmo 1.10 *Búsqueda dicotómica.*

En el Algoritmo 1.10 se puede ver el código de una búsqueda dicotómica. Se trata de encontrar la posición que ocupa un número real dentro un vector ordenado. Esta función recibe el vector de reales, los índices izquierdo y derecho de los extremos en los que se centra la búsqueda, y el elemento a buscar.

Para hacer el cálculo de la eficiencia del Algoritmo 1.10 habrá que identificar los tres parámetros que nos permitan utilizar el Teorema Maestro II. En este caso, sin embargo, primero nos hace falta definir el tamaño de los datos n, ya que no aparece en el código de una manera tan explícita como en cualquiera de los ejemplos anteriores. Así pues, antes de entrar en la identificación de los tres parámetros del Teorema 1.2 hay que dejar bien claro a qué le llamamos el tamaño de los datos de una instancia. Esta definición ha de partir de los parámetros de la función recursiva. Para el caso de la búsqueda dicotómica, diremos que $n = d - i + 1$, ya que intuitivamente, asociamos el tamaño de la instancia al tamaño del espacio de búsqueda. Una vez establecida esta n procedemos a utilizar el Teorema Maestro II para las recursividades divisoras.

Como se ha venido diciendo, a es el número de llamadas recursivas en ejecución. Eso significa el número de llamadas que realmente se realizan en el flujo de ejecución del algoritmo. Así pues, a pesar de que la llamada aparezca dos veces en el código, el parámetro a vale 1, ya que en ningún caso se ejecutarán las dos. Por otra parte, teniendo en cuenta la definición de n que acabamos de hacer, no resulta difícil darse cuenta de que entre llamadas sucesivas se reduce a la mitad. Es decir, $b = 2$. Y finalmente, la k. Llamadas recursivas aparte, el número de operaciones elementales es constante (puede ser seis o siete, pero constante en definitiva) y por tanto, $\Theta(1)$. Eso es, $k = 0$. Luego, entramos al Teorema 1.2 por el caso $a = b^k$, ya que $1 = 2^0$, lo cual nos clasifica el tiempo de la búsqueda dicotómica en $\Theta(n^k log(n))$, que es

$$T(n) = \Theta(log(n)).$$

En este capítulo se ha presentado la notación asintótica. En síntesi, podemos recordar que $f \in \Theta(g)$ si existe el límite $lim\ f/g$, excluyendo el caso que sea cero. Si es cero, entonces $f \in O(g)$, o sea que g acota por encima f. Si el límite no existe, o sea, si $lim\ f/g = \infty$, entonces $f \in \Omega(g)$.

Capítulo 2

Estructuras de Datos

Este capítulo, más que ser una descripción global de las estructuras de datos habitualmente usadas en los programas informáticos, es un análisis de aquéllas que por sus operaciones internas tienen cierto interés algorítmico. Así, se supone al lector un conocimiento más o menos profundo de las estructuras de datos contenedoras de elementos que se diferencian entre ellas por las operaciones más comunes: Listas, pilas, colas, etc...

Se abre el capítulo con una introducción que va más allá de lo que se podría pensar para un texto dedicado a las estructuras de datos. En ella se habla de temas mucho más filosóficos que el lector podría considerar que no tienen nada que ver con la programación informática. Sin empbargo, el concepto de estructura de datos pertenece en última instancia a conjuntos mucho más abstractos de conceptos, y siempre se puede empezar hablando del universo cuando uno quiere hablar de la cocina de su casa. Es, en definitiva, una disertación que conviene mantener en mente cuando nos dedicamos a construir teorías, aunque lo que después queda recogido en el capítulo toque mucho más de pies en el suelo.

Así pues, se aconseja al lector que pretenda extraer un conocimiento práctico del capítulo, que se salte la introducción, y solamente sea leída por aquéllos que tengan curiosidad en saber qué fundamento tienen estas herramientas que llamamos estructuras de datos.

Después se presentan los diccionarios, las colas de prioridad, y las particiones. Son estructuras conceptuales, lo bastante genéricas como para poder ser implementadas de diversas formas. Por esto, hablaremos de implementaciones de los diccionarios o de las colas de prioridad, que a su vez serán implementadas por diferentes vías. Implementaciones de implementaciones. Resulta retorcido. Adelante, las cuestiones se irán aclarando.

2.1 Introducción

Hay una primera sección en la que se habla de lógica. No tiene demasiada relación con lo que se explica en el capítulo en general, pero está aquí para dejar bien claro que las estructuras de datos son definidas por consenso. Podrían ser de otras maneras ya que en definitiva son herramientas que a lo largo de la historia de la programación de computadoras se ha ido observando que resultarían útiles. En la segunda parte de esta introducción hay una breve disertación sobre el conocimiento deductivo y el conocimiento empírico. Lo que se pretende con estas dos secciones es situar el resto del capítulo donde le corresponde. Es decir, dentro de las teorías abiertas.

2.1.1 Lógica

En el mundo de la lógica matemática se utilizan intensivamente dos conceptos que en principio son contrarios. Los llamamos *cierto* y *falso*. En particular, en algún área de la lógica matemática se habla de *sistemas formales* como aquellos sistemas de deducción en los que se parte de un conjunto de certezas o verdades que llamamos *axiomas* (o certezas atómicas), y también de un conjunto de reglas para operar con estos axiomas, y así formar *teoremas*, es decir, certezas demostrables a partir de los elementos atómicos en aquel sistema formal. Yendo un poco más allá, podemos considerar una *teoría*, que también podemos llamar *lenguaje*, como un conjunto de teoremas. No es difícil intuir que el análisis matemático o los lenguajes humanos son sistemas formales.

Es más. Desde este punto de vista, se entiende que cada objeto a nuestro alrededor, un teléfono, un zapato o un vaso de agua, contienen sus propios lenguajes. Para centrarnos en un solo ejemplo consideremos el teléfono. Un teléfono es una teoría. Sus axiomas son los botones, el micrófono, el altavoz, y la línea telefónica. Es decir, lo común a todos los teléfonos. El lenguaje del teléfono nos permite marcar un número. Nos permite colgar y cambiarlo de lugar. Incluso nos permite tirarlo bien lejos o chafarlo de un martillazo. Todo aquello que podamos hacer con un teléfono cualquiera forma parte de su lenguaje. Con un vaso de agua ocurre igual. Con el lenguaje de un vaso de agua puedo expresar el mensaje de rellenarlo, de beberlo, o de derramarlo si intento llenarlo más allá de su capacidad. Lo que también parece bastante claro es que con un vaso de agua no puedo marcar un número de teléfono, igual que no llenaría de agua un teléfono. Esto es, dentro de cada teoría se define un conjunto de teoremas que viene a ser el conjunto de mensajes formulables con el lenguaje que es la teoría.

Entonces, podemos clasificar las teorías en tres grandes grupos: Abiertas, categóricas e inconsistentes.

Una teoría abierta es una teoría interpretable. También se puede llamar

incompleta. Es aquella teoría que puede alimentarse de elementos que no forman parte de su definición para continuar formulando nuevos teoremas. Los tres ejemplos del párrafo anterior pertenecen a este grupo, ya que un teléfono puede ser rojo o azul y aún así seguir siendo un teléfono. El color no forma parte de la teoría del teléfono ya que no forma parte de su definición, y por tanto, podemos enriquecer la teoría formando nuevas teorías. La teoría de los teléfonos esféricos o planos... En otras palabras, las teorías abiertas dicen lo que dicen y no van más allá.

De teoría categórica solamente tenemos un ejemplo. Se trata del análisis matemático. El propósito de una teoría categórica es mucho más ambicioso que en el caso anterior. Ahora pretendemos que, para cualquier cuestión formulable, la teoría nos responda si es cierta o falsa. Actuamos como si las cosas que podemos enunciar fueran todas ciertas o falsas, y la utilidad de la teoría es precisamente colocar cada cuestión en uno de los dos lados.

Y tenemos las teorías inconsistentes finalmente. Éstas no parece que tengan ninguna utilidad. Simplemente es el espacio donde van a morir las teorías. En ellas hay algún error estructural que abre brechas por donde la falsedad se cuela en el conjunto de las verdades demostrables produciendo contradicciones. Por eso también se pueden llamar teorías contradictorias.

Conviene reflejar que la vida de un ser humano es una teoría que pasa por los tres estados a lo largo del tiempo. Todo el mundo puede ser presidente de los Estados Unidos en el momento de nacer. A medida que se va creciendo con verdades subjetivas vamos formando la teoría que es cada persona. Finalmente, todos lo sabemos todo, aunque por entonces ya no sirve para nada. También hay una analogía entre una partida de ajedrez cuando finalmente se pierde y los tres conjuntos de teorías, ya que cuando jugamos una partida comenzamos en un estado correspondiente a una teoría abierta. Las primeras jugadas perfectas son varias posibles. Más adelante, aunque quizás con no demasiada claridad, se puede distinguir una jugada donde se inicia nuestra derrota. Y una vez pasado ese punto, ya no tenemos remedio.

2.1.2 Conocimiento

La actividad de estudiar tiene dos grandes vertientes. Por un lado hay un tipo de estudio eminentemente observador. La información en ese caso transita de la circunstancia al individuo. Se trata del estudio en el que se pretende construir un modelo de algún fenómeno que constituye precisamente el objeto del análisis. Y por otra parte hay otro tipo de estudio que es eminenmente creativo. En este caso la información mana del individuo hacia el exterior. Normalmente, este tipo de estudio creativo es posterior en el tiempo que aquél otro. Se trata de proveernos de instrumentos que después de la observación, la persona que estudia ha creído que le podrían ser útiles para sus propósitos.

Está claro que el concepto de estructura de datos corresponde a las teorías abiertas, y que pertenece a esta segunda clase de estudio.

Con esta breve introducción se pretende dejar claro que lo que seguidamente se expone es así porque así se ha creído la necesidad de ser. No vamos a estudiar cosas difíciles, sinó cosas útiles.

2.1.3 ¿Por qué nos inventamos estructuras de datos?

Cuando, con el tiempo y la experiencia, observamos que hay colecciones de elementos con las que necesitamos realizar ciertas operaciones de manera frecuente, entonces pensamos que debe existir alguna manera de guardar los elementos en esas colecciones para agilizar en la medida de lo posible esas operaciones habituales.

Es decir, después de llevar tiempo haciendo programas informáticos y de observar que hay acciones comunes en distintos ámbitos que requieren más tiempo del deseable, decidimos crear estructuras especializadas para agilitzar esas acciones.

De las estructuras de datos que en este texto nos disponemos a analizar, tan sólo nos interesan las que haya alguna idea digna de observar en su implementación. En ellas, o más concretamente, en sus implementaciones habrá un interés algorítmico que las hace merecedoras de ser analizadas en un contexto como el que aquí se hace.

Son tres. Nos disponemos a estudiar tres estructuras de datos: Los diccionarios, las colas de prioridad, y las particiones.

Los diccionarios se caracterizan por ser estructuras de datos especializadas en almacenar (cosa común a todas las colecciones) y en buscar. Nos inventamos los diccionarios porque necesitamos alguna estructura capaz de encontrar un elemento entre una colección de la forma más rápida que sea posible.

Les colas de prioridad se caracterizan por ser estructuras de datos especializadas en almacenar, claro, y en obtener el máximo. Obtener el máximo. Está claro que los elementos contenidos en las colas de prioridad, tienen asociado un valor numérico. A esta parte del elemento le llamaremos prioridad.

Las particiones son estructuras de datos que nos permiten implementar las clases de equivalencia. Como ya se ha dicho en el capítulo anterior, las relaciones de equivalencia nos definen particiones de universos de individuos. Es por tanto interesante disponer de una estructura de datos que a la larga, si no la tuviéramos, la echaríamos de menos.

Y ya está, no hay más. Vamos a estudiar tres tipos de estructuras de datos que las tres guardan colecciones de elementos. En los tres casos estos elementos están formados por dos partes diferenciables. Un campo funcional y un apéndice descriptivo con el que no realizaremos ninguna clase de operación. Respeto al campo funcional, serán: En los diccionarios para poderse buscar, la clave. En las colas de prioridad para poder establecer un orden numérico, la prioridad. Y para las particiones, para poderse agrupar, también lo llamaremos clave.

En definitiva, hay que entender que establecemos definiciones de estructuras de datos para agilizar operaciones habituales y comunes en diferentes ámbitos.

Quizás, con el tiempo, surgirán otras estructuras especializadas en otras operaciones. Imaginemos, por ejemplo, una colección de elementos de tal naturaleza que su utilidad aparezca en el momento en que formen parejas. Imaginemos elementos con formas geométricas como piezas de puzzle que tuvieran un atributo, o una propiedad o valencia, que nos indicara con cuáles otros elementos pueden formar parejas más o menos adecuadas. Entonces podríamos estudiar la manera de almacenar estos elementos que nos resultara ágil en el momento de extraer una pareja útil... y como éste, nos podríamos inventar muchos otros ejemplos que si de momento no hemos implementado es porque no ha surgido la necesidad de utilitzarlos.

2.2 Diccionarios

¿Cómo pondríamos todas las palabras posibles, existentes y no existentes, en una secuencia ordenada?. Si quisiéramos hacer un diccionario con todas las palabras posibles, parece claro que la primera palabra sería la "a". ¿Y la segunda?. No os confundáis. Quien piense que la segunda sería la "aa", entonces deberá pensar que la tercera sería la "aaa". Y por tanto, nunca llegaría a la "b". Así pues, atención. Si queremos un diccionario con todas las palabras posibles las deberíamos ordenar por longitud de palabra como orden principal. Y, dentro de todas las palabras de una misma longitud, entonces sí que podríamos utilitzar el orden alfabético tradicional. Es decir, la segunda palabra sería la "b", y la "aa" iría inmediatamente después de la "z".

En definitiva, trataríamos las letras como si fueran cifras de un sistema de numeración en base 26, el número de letras del alfabeto. Quien no recuerde la forma polinómica de un número que atine un instante en que el número 723 en base diez significa $7*10^2 + 2*10^1 + 3*10^0$. Con esta regla, pero en base 26 en lugar de 10, sería sencillo saber en qué posición hay cada palabra. Sería fantástico saber en qué página del diccionario está la palabra que buscamos antes de abrirlo. Por ejemplo, "casa" estaría en la posición $3*26^3 + 1*26^2 + 19*26^1 + 1*26^0$ ya que la "c" es la tercera letra, la "a" la primera, y la "s" la decimonovena del alfabeto. Esta suma es igual a 53899. Ya se ve que este número es muy grande. Si quisiéramos tener todas las palabras de hasta diez letras, necesitaríamos un vector de $26^{11} - 1$ posiciones. Eso, en decimal tendría 16 cifras (recordad, 11

veces el logaritmo en base 10 de 26). Es decir, unos diez mil billones con b. Es demasiado grande. Aún así, si pudiéramos tenerlo en memoria sería fenomenal. En cada posición del vector podríamos guardar la definición de la palabra, y el tiempo que tardaríamos en saber una definición cuando tuviéramos una palabra para buscar sería $\Theta(1)$, ya que tan sólo deberíamos desplazarnos desde el inicio al lugar buscado, y eso se haría con una suma. No obstante, no podemos tener un vector tan grande en ningún ordenador.

Para aquellos lectores que no conozcan demasiado la historia de la programación de ordenadores, que se sepa que utilizar memoria dinámica significa tener instrucciones, en el lenguaje de programación, que nos permitan pedir memoria en tiempo de ejecución. Antes no era así. Los programas en fortran o en pascal comenzaban, justo bajo la cabecera del código fuente, por declarar la cantidad de memoria que se disponían a utilizar. Esta memoria se reservaba en tiempo de compilación, y en ningún caso se podía exceder en tiempo de ejecución. Todo esto viene a cuento porque los diccionarios, a los que también llamamos tablas de símbolos, son unas estructuras de datos utilizadas por los compiladores. Dicen que quien no conoce su historia está condenado a repetirla, o que quien pierde los orígenes pierde la identidad. Antes de que existiera el lenguaje java, antes de la programación orientada a objetos, antes del lenguaje C e incluso antes de utilizar memoria dinámica en la programación con lenguajes de alto nivel, ya existían los diccionarios. En todo este libro hay pocos puntos, o quizás ningún otro, en los que aflore de una manera tan clara la historia de la programación como en este tema, las tablas de símbolos.

Bien, un diccionario es una recopilación de descripciones asociadas una a una a un conjunto de claves. En pro de la sencillez, y de no requerir tanto conocimiento del lenguaje C++, nos limitaremos a los tipos de datos primitivos. Supondremos que las claves de los elementos son de tipo entero, y las descripciones, vectores de carácteres de una longitud máxima fijada. O sea, consideraremos un diccionario como una recopilación de elementos con la estructura mostrada en el Algoritmo 2.1.

```
#define MAXLEN 1024
struct elemento
{
      int clave;
      char descripcion[MAXLEN];
};
```

Algoritmo 2.1 *Estructura de datos para los elementos del diccionario.*

A la hora de estudiar las implementaciones para los diccionarios, vaya por delante que en todas ellas, excepto cuando implementemos un diccionario en un vector, utilizaremos la estructura del Algoritmo 2.1 para los elementos.

Una implementación más genérica de esta estructura utilizaría plantillas para poder parametrizar los dos tipos de datos que contiene. En ese caso, el tipo parametrizado para el campo *clave* debería incorporar un operador de ordenación, "<", que retornara un booleano.

Aquí no establecemos ningún tipo de restricción para las descripciones. En cambio, exigimos que las claves sean ordenables. Y por tanto comparables. En rigor, si dos elementos del diccionario son diferentes, tienen la clave diferente. Eso significa, de rebote, que podemos identificar cada uno de los elementos del diccionario por su clave.

Ahora bien, como estructura de datos, la definición de un diccionario contempla además su operación básica.

Definición 2.1 Diccionario. *Estructura de datos contenedora de ítems con claves especializada en la operación de buscar.*

El problema que nos interesa resolver en esta sección es: ¿Cómo nos guardamos un diccionario para agilizar las búsquedas?. Lo ideal sería tener un vector muy grande. Pero tan grande que no cabría en memoria. Además, por otra parte, no sólo queremos indexarlo utilizando números naturales, sinó también con palabras alfabéticas.

Hay una categorización para los diccionarios según las operaciones que permiten:

- *estático:* Cuando solamente permite ser creado y consultado. Es decir, se llena de contenido cuando se crea y se ha de entender como una estructura de sólo lectura con tan sólo las operaciones de crear y buscar (utilizamos consultar y buscar como sinónimos).

- *semidinámico:* Cuando, además, permiten insertar.

- *dinámico:* Cuando permite todo lo anterior y además añade la posibilidad de eliminar.

En adelante, estudiaremos diversas posibilidades para la implementación de los diccionarios. Y para cada una de ellas, analizaremos la eficiencia de las operaciones más comunes: crear, insertar, buscar y eliminar.

A lo largo de estas secciones, notaremos por K el conjunto de todos los valores posibles de clave. Es decir, el valor de una clave es un elemento del conjunto K. Y se supone que $|K|$ es un número grande.

2.2.1 Implementaciones Sencillas

Con implementaciones sencillas para los diccionarios nos referimos a dos estructuras de datos: Una de estática, el vector, y otra de dinámica, la lista. Aunque sólo sea para hacer un repaso y para poner en práctica los términos definidos en el capítulo anterior, echemos un vistazo a estas dos posibilidades.

Vector

El concepto informático de *vector* viene importado del mismo concepto matemático. En inglés, en cambio tienen dos palabras diferentes, *vector* y *array*. Al traducirse al castellano por la parte de Méjico, utilizaron y aún se hace uso, de los vocablos *vector* y *arreglo*. Aún así, en el castellano ibérico desde los inicios se utilizó mayoritariamente sólo la palabra *vector*, aunque también *arreglo* de forma más residual. Este tema no está exempto de polémica.

Hay quien nunca ha aceptado el término *arreglo* arguyendo que esencialmente es el mismo concepto de vector, y por tanto no es necesaria ninguna palabra adicional.

No obstante, la diferencia entre los dos conceptos es que *vector* se refiere a una secuencia indexable de números, mientras que *array*, o *arreglo*, se utiliza de manera más genérica como secuencia indexada de cualquier tipo de información siempre que todos los elementos de la secuencia tengan una misma estructura. Nosotros, en español, le llamaremos *vector* que también, como se ha visto, tiene la ventaja de ser internacional.

En relación a los diccionarios, el vector es probablemente la implementación más sencilla que nos podamos imaginar. En la práctica ya se ve que en el fondo tan sólo es una idea teórica, ya que por el ejemplo de uso que se ha hecho en la introducción de esta sección, sabemos que su implementación para colecciones grandes de claves no es factible. Esta implementación sólo será viable, pues, si el universo de claves posibles K, es un conjunto con muy poquitos elementos. Este es el caso del Algoritmo 2.2, donde se implementa la estructura *diccionario_vector*.

Como ya se ha dicho, esta implementación sólo tendría sentido si K pudiera ser mucho mayor. Tal como está, resulta insuficiente por el hecho de poder guardar solamente mil claves. Además, las claves deben ser enteras y no de ningún otro tipo, y su dominio, ser exactamente del 0 al 999. En cualquier caso, si pudiéramos utilizar la misma idea para una K mucho mayor (como se ha dicho antes, $26^{11} - 1$), entonces esta sencilla estructura ya nos permitiría trabajar con tipos de claves más útiles.

```
#include "elemento. h"
#define K 1000
class diccionario_vector
{
      char T[K][MAXLEN];
      int n;
public:
      diccionario_vector()
      {
            n = 0;
            memset(T,K*MAXLEN,0);
      }

      bool insertar(elemento e)
      {
            if (0<=e.clave && e.clave<K) {
                  strcpy(&T[e.clave][0],e.descripcion);
                  n++;
                  return true;
            }
            return false;
      }

      bool buscar(element& e)
      {
            if (0<=e.clave && e.clave<K) {
                  strcpy(e.descripcion,&T[e.clave][0]);
                  return true;
            }
            return false;
      }
};
```

Algoritmo 2.2 *Implementación estática de un diccionario en un vector.*

Analicemos la eficiencia de las operaciones más frecuentes cuando implementamos un diccionario con el código mostrado en el Algoritmo 2.2:

Por un lado, para la creación tendríamos $\Theta(K)$, en cualquier caso. O sea, independientemente del número de elementos que vayamos almacenar. Eso, suponiendo que el tiempo de reservar el espacio necesario para un elemento fuera $\Theta(1)$. Cuando las eficiencias no dependen del número de elementos, o sea $\Theta(1)$, y por tanto no dependen de los datos de entrada, entonces nunca habrá distinción de casos mejor, medio, o peor. De hecho, es un poco absurdo decirlo, ya que cuando se crea una estructura de datos para almacenar una colección de elementos, normalmente no se sabe cuántos elementos será necesario almacenar en esta colección, y por tanto, las eficiencias de creación siempre serán $\Theta(1)$,

independientemente del número de elementos.

Por otro lado, con esta estructura en particular ni siquiera es necesario distinguir entre mantenerla ordenada o no, ya que lo está por definición. Eso es, tanto insertar, como buscar, como eliminar, requerirán $\Theta(1)$ en cualquier caso, o sea, independientemente de los valores que se inserten, busquen, o eliminen.

Sin duda, en un sentido funcional ésta sería la más deseable de las implementaciones. De todas formas, el hecho de requerir un espacio $\Omega(K)$ la convierte del todo en inadmisible. En ningún caso, nunca jamás, podemos aceptar estructuras que dependan del contenido de los datos, es decir, de los valores del conjunto K para los diccionarios.

Lista

La implementación de un diccionario en una lista utilizará la estructura *nodo* que se muestra en el Algoritmo 2.3.

Técnicamente, es interesante observar que cada envoltorio que estamos haciendo a los datos tiene una finalidad bien definida. En primer lugar, con la definición del tipo *elemento* del Algoritmo 2.1, estamos envolviendo los datos en una estructura que permite su identificación. Ahora, en un segundo nivel, envolvemos esos elementos ya identificables para poderlos encadenar.

```
struct nodo {
    elemento e;
    nodo* siguiente;
};
```

Algoritmo 2.3 *Estructura básica de la lista.*

Y más filosóficamente, esta estructura de cebolla es como la de la mente humana. En su núcleo tiene la conciencia del yo, y el instinto de supervivencia, conceptos propios del individuo, y en zonas más periféricas del cerebro, conceptos como el lenguaje, de naturaleza más colectiva.

La estructura *lista* es, como se ha dicho, una estructura dinámica. Eso significa que utiliza una llamada al sistema operativo para pedir memoria, *new*. Lo hace cada vez que se inserta un nuevo elemento, y no tiene otra limitación que la memoria total disponible en el sistema. Esto es una ventaja. Y una responsabilidad, también.

Hasta antes de que existiera la memoria dinámica, a finales de los ochenta,

un programador informático en lenguajes de alto nivel, no tenía ninguna forma muy clara de poder incidir en los procesos que corrían juntamente con el suyo en un sistema multitarea. A partir del uso extendido de las solicitudes de memoria, ya se tiene un punto de interferencia importante entre procesos.

Así pues, hay que tener presente que cuando se pide memoria, se libera. Ya se ha dicho que el problema más grave que tienen los programas de hoy día son los punteros descontrolados. Y es de muy mal gusto por parte del programador no tener bajo control la cantidad de recurso que se le ha concedido.

Porque está bien claro que un sistema operativo no tiene por qué ser capaz de controlar ese extremo, ya que para hacerlo, debería intervenir en multitud de ocasiones ralentizándose de forma notable. El sistema no puede saber qué hace un programa con la memoria que le ha concedido a partir de alguna dirección, y si esta dirección inicial de la sección de memoria otorgada al proceso, por la razón que sea, es machacada por el mismo proceso, no siempre es fácil recuperarla, en lenguaje C. En java, en cambio, hay la máquina virtual que ejerce de intermediaria. Y es ésta, la razón de su existencia: Apuntar-se cuánta memoria ha concedido a cada proceso y a partir de dónde. Así, le resulta fácil recuperarla cuando el proceso muere. Eso ahorra los *delete*s y muchos problemas en java, a costa de ir más lento.

O sea, que la evolución de la programación informática ha pasado de no poder reservar memoria, fortran o pascal, a poderlo hacer bajo el compromiso de liberarla, C, pero como los programadores de todo el mundo éramos unos descontrolados, finalmente vino el java a poner orden, prohibiéndonos descontrolar la memoria que se nos había dejado.

El Algoritmo 2.4 muestra la implementación de un diccionario en una lista no ordenada. Su eficiencia para la creación es $\Theta(1)$ en cualquier caso, o sea, independientemente del número de elementos que vayamos a almacenar.

Como se ha apuntado más arriba, no tiene demasiado sentido hablar de eficiencias en la creación, ya que el tamaño de los datos en todos los algoritmos de este capítulo se corresponden con el número de elementos de la colección implementada en la estructura. Por tanto, como en el momento de la creación acostumbramos a no tener ningún elemento, no podemos poner la eficiencia en función de n.

```
#include "nodo.h"
class diccionario_lista {
      int n;
      nodo* primero;
public:
      diccionario_lista() { n=0; primero = NULL; }
      ~diccionario_lista() {
            nodo* actual = primero;
            for (int i=0; i<n; i++) {
                  nodo* siguiente = actual→ siguiente;
                  delete actual;
                  actual = siguiente;
            }
      }
      void insertar(elemento e) {
            if (n==0) {
                  primero = new nodo;
                  primero→ e = e;
                  primero→ siguiente = NULL;
            } else {
                  nodo* actual = primero;
                  for (int i=0; i<n-1; i++) {
                        actual = actual→ siguiente;
                  }
                  actual→ siguiente = new nodo;
                  actual = actual→ siguiente;
                  actual→ e = e;
                  actual→ siguiente = NULL;
            }
            n++;
      }
      bool buscar(elemento& e) {
            nodo* actual = primero;
            while (actual && actual→ e.clave != e.clave) {
                  actual = actual→ siguiente;
            }
            if (actual != NULL) {
                  e = actual→ e;
                  return true;
            }
            return false;
      }
};
```

Algoritmo 2.4 *Implementación dinámica de un diccionario en una lista.*

Hagamos ahora una inmersión en un análisis de eficiencia interesante. La

cuestión que nos prenguntamos es si vale la pena ordenar la lista. La razón fundamental por la cual se ordenan las cosas es para poderlas buscar más tarde. La búsqueda dicotómica, como se ha introducido en la Sección 1.6.5, sólo puede realizarse en vectores ordenados. Respeto las listas, no obstante, una razón de peso para no ordenarlas es que no haya acceso directo a los elementos, y por tanto, las búsquedas seguirían siendo lentas aunque las ordenáramos. Independientemente de esto, observad por otra parte que para ordenarlas no sería necesario desplazar los elementos sinó gestionar correctamente los apuntadores.

Para poner fondo, supondremos que las claves que se pueden insertar o buscar siguen una distribución de probabilidad uniforme. En otras palabras, que pueden tener cualquier valor de K con la misma probabilidad. Distinguiremos casos. En este análisis usamos la palabra *valor* para referirnos al *valor de la clave de los elementos*.

- Si la lista está ordenada, para insertar tardaremos...

caso peor: $\Theta(n)$, o sea, cuando el valor a insertar es mayor que todos los otros valores del diccionario.

caso medio: $\Theta(n/2)$, o sea, la totalidad de los valores a insertar, tantas veces irán a parar antes como después de la mitad de los valores del diccionario (distribución uniforme).

caso mejor: $\Theta(1)$, o sea, cuando el valor a insertar es menor que el más pequeño de los valores del diccionario. Fijaos que, el mejor caso no depende de la cantidad de elementos que haya en el diccionario. Claro, porque para insertarlo, sólo es necesario compararlo con el más pequeño y basta, independientemente de cuantos elementos haya tras él.

- Si la lista está desordenada, para insertar tardaremos $\Theta(1)$ en cualquier caso, o sea, no dependerá nunca del número de elementos.

Recordad que por definición $\Theta(n/2) = \Theta(n)$. El hecho de poner las dos expresiones es para dar un poco más de precisión involucrando la constante oculta.

En un diccionario implementado en una lista ordenada, la búsqueda tardará $\Theta(n/2)$ en el caso medio y $\Theta(n)$ en el caso peor. Como ya se ha dicho, esto es debido a que en una secuencia ordenada la búsqueda puede resolverse con $\Theta(log(n))$ cuando se dispone de acceso directo a los elementos. Si la búsqueda se hace en una lista no ordenada, está claro que será $\Theta(n)$.

Finalmente, para la eliminación tenemos en todos los casos el mismo comportamiento que para la búsqueda, puesto que eliminar es lo mismo que buscar respeto al esfuerzo computacional (más la $\Theta(1)$ que representa borrar el elemento cuando ya lo hemos encontrado).

La implementación de un diccionario en una lista es factible. Eso ya la coloca mejor que la implementación en un vector de la sección anterior. Con la lista, al menos, ya podemos manejar grandes conjuntos de claves. De todas formas, en definitiva, está claro que si estamos buscando una estructura de datos que nos resuelva las búsquedas ágilmente, esta implementación tampoco es la solución.

2.2.2 Tablas de Dispersión: Hashing

La estructura de datos que permite implementar la idea de vector enormemente grande son las tablas de dispersión, o *hashing*.

Tocando de pies en el suelo, tenemos un vector con un número razonable de posiciones, M. Por otro lado tenemos un conjunto muy grande de claves, K. Se trata de tener una función que dada una clave retorne un índice entre 1 y M, donde guardar el elemento. A esta función le llamaremos función de hash, o función de dispersión.

Definición 2.2 Función de Hash (O función de dipersión). *Es una transformación que toma elementos del conjunto de claves posibles, K, y nos retorna un índice del conjunto de índices posibles* $[1..M]$.

Expresado en términos formales,

$$h : \underset{k}{K} \rightarrow \underset{h(k)}{[1, M]}.$$

En principio, $|K|$ es mucho mayor que M. En otras palabras, la cantidad de claves posibles es mucho más grande que el número de entradas disponibles en el vector.

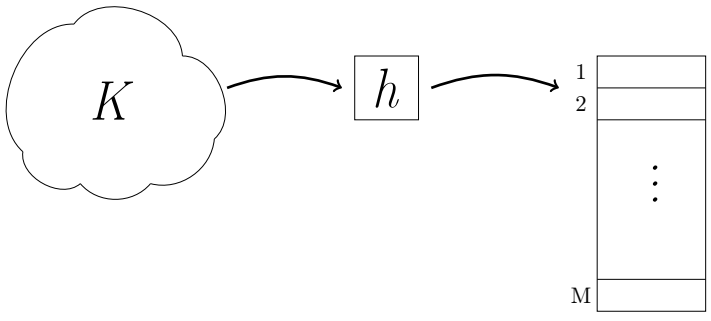


Figura 2.1: *Esquema gráfico del funcionamiento de una función de dispersión,* h(k).

En la Figura 2.1 se puede observar una idea gráfica del funcionamiento de las tablas de dispersión.

La función de dispersión pues, es exhaustiva. O sea, diferentes valores de clave pueden ser transformados por $h(k)$ al mismo índice i, $1 \leq i \leq M$. Una buena función de hash debería evitar que esto se produjera.

Precisamente por la antigüedad que tienen las tablas de dispersión como implementación de las tablas de símbolos, nos encontramos con una gran cantidad de léxico. Llamaremos *sinónimos* a dos elementos con claves diferentes que una vez transformadas por la función de dispersión coinciden en el mismo valor de índice.

$$k_1, k_2 \in K \text{ sinónimos} \Leftrightarrow k_1 \neq k_2 \wedge h(k_1) = h(k_2).$$

Para el caso, se utiliza la palabra *colisión*. O sea, cuando introducimos dos elementos sinónimos en un mismo diccionario se produce una colisión. Una buena función de dispersión, pues, debería evitar las colisiones en la medida de lo posible. Para esto, hay que tener en cuenta la probabilidad de que se produzcan ciertos valores de clave. Nos interesa que la probabilidad sea tan uniforme como sea posible. O, en otras palabras, que la entropía sea máxima.

Ejemplos de funciones de hash podrían ser la última cifra del número del pasaporte, o la primera letra del apellido (bien, una transformación numérica asociada, como el código ASCII de la letra). En cambio, si usáramos la primera cifra del pasaporte tendríamos más colisiones, e igualmente sucedería con la última letra del apellido. Esto en España es así porque el número de pasaporte coincide con el del DNI. Y no hay DNIs que empiecen con 8 ni con 9, por tanto la probabilidad no sería uniforme. De igual manera se puede intuir que así como las iniciales de los apellidos pueden ser cualquier letra del alfabeto, difícilmente encontramos apellidos que acaben en "j", o en "q".

Observad que en cualquier caso, cuando un nuevo ítem es introducido en la colección, siempre tendremos que guardarnos la clave completa inicial con el elemento, ya que cuando se produce una colisión acabaremos comparando las claves enteras. Es decir, antes de haberse transformado con la función de dispersión.

Los diccionarios implementados en tablas de dispersión han de tener un tratamiento de colisiones. Eso es, una forma prestablecida de reubicar un elemento cuando el índice que le ha correspondido por su clave ya ha sido ocupado previamente por algún sinónimo. Hay dos técnicas de tratamiento de colisiones que analizaremos seguidamente. Se llaman *direccionamiento abierto* y *encadenamiento separado*.

Por otro lado, en cualquiera de los dos casos, diremos *factor de carga* al número n/M, siendo n el número de claves efectivamente introducidas en el diccionario, y por tanto, desconocido a priori, $n = n(t)$. El factor de carga adquiere trascendencia al analizar la eficiencia. A parte de crear el diccionario, que ya que M no depende de n, la creación es $\Theta(1)$, las otras tres operaciones serán $\Theta(n/M)$, o lo que es lo mismo, los tiempos de realizarlas crecen siempre

como el factor de carga.

Direccionamiento abierto

La técnica más antigua de tratamiento de colisiones en las implementaciones en tablas de dispersión recibe el nombre de direccionamiento abierto. Es la única estructura de datos de este libro que no utiliza memoria dinámica. Como ya se ha dicho, eso se debe a que tiene más años el hashing que la memoria dinámica. El nombre de direccionamiento abierto proviene del hecho de que en última instancia, cualquier clave puede ser guardada en cualquier posición del vector.

En su versión más simple, la regla básica de esta técnica consiste en, cuando se produce una colisión, buscar secuencialmente la primera entrada libre en el vector.

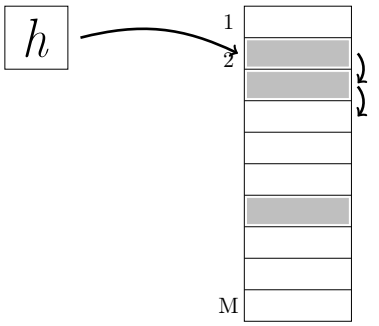


Figura 2.2: *Direccionamiento abierto.*

En la Figura 2.2 las posiciones sombreadas del vector significan posiciones ya ocupadas. Pongamos por ejemplo que estemos guardando personas a partir de su DNI. Además, supongamos que la función de dispersión tiene la forma $h(\text{DNI}) = \text{DNI} \bmod 10$. Es decir, que para conocer la posición del vector donde, en principio, tiene que ir un DNI concreto, sólo nos fijamos en la última cifra. Supongamos también que hasta ahora tenemos los DNIs 46123212, 34512323, y 46541447. En este punto hay que introducir una nueva alta correspondiente al número 43175492. El nuevo elemento debería ir a la posición 2 del vector, pero como ya está ocupada por el 46123212, vamos buscando secuencialmente hasta encontrar la primera celda libre, que en el ejemplo es la 4. Luego, el último DNI insertado iría a ocupar esta celda.

En este ejemplo se ha visto el caso más sencillo del tratamiento de colisiones con direccionamiento abierto. En general, de una manera más formal, para el tratamiento de colisiones definimos lo que llamaremos función de redispersión, $H(k, i)$.

Definición 2.3 Función de Redispersión. *Transformación que dada una clave $k \in K$, y su índice obtenido con la función de dispersión $i = h(k) \in [1, M]$, retorna un nuevo índice $H(k, i) \in [1..M]$.*

En otras palabras, una función de redispersión es,

$$H : \underset{k,i}{K \times [1..M]} \rightarrow \underset{H(k,i)}{[1..M]}.$$

A menudo se hace mención del hecho de que, por definición, tanto $h(k)$ como $H(k, i)$ tienen que ser $\Theta(1)$. De todas formas, teniendo presente que cuando hablemos de diccionarios el tamaño de los datos es el número de elementos del diccionario, que estas funciones tengan que ser $\Theta(1)$ tan sólo significa que no podemos tener en cuenta los otros elementos del diccionario en el momento de asignar un índice a una nueva clave. Cosa que por otra parte, ya queda suficientemente clara.

Una vez introducido el concepto de redispersión, aún hay algunos términos del léxico asociado a las tablas de dispersión que conviene, almenos, mencionar. Para esto, volvemos a hacer referencia al ejemplo mostrado en la Figura 2.2 de la página 70. Observando el dibujo se puede contar que M es igual a 10. Entonces, tal como se ha indicado, la función de dispersión se queda con la última cifra del carnet de identidad (en rigor, hay un desplazamiento de una unidad para hacer que el índice del vector comience en 1).

Los parámetros definidos para las tablas de dispersión que implementan el tratamiento de colisiones con la técnica de direccionamiento abierto para el ejemplo de la Figura 2.2 representan:

- K = conjunto de los DNIs.
- $h(k) = 1 + k \bmod 10$.
- $H(k, i) = 1 + (i + c) \bmod 10$, siendo para el caso concreto $c = 1$.

El parámetro c recibe el nombre de *probe position*. Con este nombre se pretende reflejar que cuando una posición está ocupada, probemos a ver si la siguiente está libre. Pero, teniendo en cuenta que la posición más rápida de acceder a partir de la actual no siempre tiene que ser la siguiente, a veces se utilizan otros valores distintos de 1.

Esta consideración se origina cuando los ordenadores tenían tambores. Un tambor era una pila de discos con varios cabezales que actuaban simultáneamente, de manera que era más rápido leer una secuencia distribuida entre diferentes discos que si se almacenaba físicamente en posiciones consecutivas.

Más concretamente, la secuencia de valores índice que se prueban cuando

la celda actual está ocupada se llama *secuencia de redispersión.* Y es de pura lógica, que conviene que la dimensión del vector, M, y la probe position, c, sean números primos entre ellos para no considerar que el diccionario está lleno cuando todavía hubiera posiciones libres. La Figura 2.3 ilustra el concepto de redispersión.

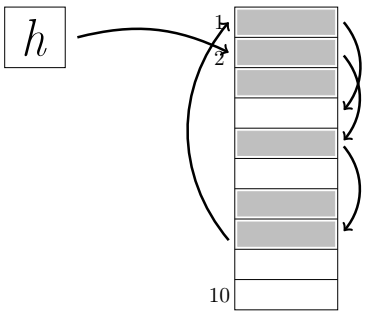


Figura 2.3: *Secuencia de dispersión para* $M = 10$ *y* $c = 3$.

En la Figura 2.3 se ilustra la secuencia de dispersión en un vector de $M = 10$ elementos, con un parámetro de posición de prueba $c = 3$. La secuencia obtenida, o *sondeo* o *exploración* linial, para una clave que nos condujera de entrada a la posición 2, sería $2, 5, 8, 1, 4$ y basta, ya que la posición 4 está libre por fin.

En el Algoritmo 2.5 se puede observar la implementación de una tabla de dispersión que utiliza la técnica de direccionamiento abierto para el tratamiento de colisiones.

Observad que en la creación hay que limpiar el vector completo ya que el valor 0 para las claves significa que son posiciones vacías.

La inserción ha de controlar que el diccionario no esté lleno, cosa que hace en su primera línea.

Además, en la búsqueda, hay un contador j que sirve para controlar que la tabla de dispersión no esté completa, ya que si no se hiciera este control, cuando se llenara el vector entraríamos en bucles infinitos ante búsquedas de elementos inexistentes.

En las dos rutinas, insertar y buscar, el bucle *while* hace el trabajo de la función de redipersión. Como se puede ver, hay la posición de prueba directamente establecida a 1.

```
#define M 1000

class diccionario_hashing_open_adressing {
      int n;
      elemento T[1+M];
      int h(int k) {return 1 + k % M;}

public:
      diccionario_hashing_open_adresing() {
          memset(T,0,(1+M)*sizeof(elemento));
          n = 0;
      }

      bool insertar(elemento e) {
          if (n==M) return false;
          int i = h(e.clave);
          while (T[i].clave != 0) i = 1 + (i+1) % M;
          if (T[i].clave == 0) {
                T[i] = e;
                ++n;
                return true;
          }
          return false;
      }

      bool buscar(elemento& e) {
          int i = h(e.clave);
          int j=1;
          while (j++ <= M && T[i].clave != 0 && T[i].clave != e.clave)
                i=1 + (i+1) % M;
          if (T[i].clave == e.clave) {
                e = T[i];
                return true;
          }
          return false;
      }
};
```

Algoritmo 2.5 *Implementación de un diccionario en una tabla de dispersión con direccionamiento abierto.*

La eficiencia de las operaciones de insertar y buscar es, como se ha dicho antes, $\Theta(n/M)$ en todos los casos. Y aunque la creación del diccionario no dependa del tamaño que tenga en cada momento y en rigor deberíamos de considerarla $\Theta(1)$, el hecho de que se deba limpiar necesariamente por cuestiones funcionales hace que tengamos en cuenta el factor $\Theta(M)$ que representa.

Casi ni hace falta decir que la mejora que representa esta implementación respeto la anterior se podría cuantificar con la relación entre el factor de carga y n. Eso significa que el hasing con direccionamiento abierto va M veces más de prisa que las listas dinámicas vistas a la sección anterior, tanto en buscar como en insertar.

Ya no sólo tenemos un implementación factible, sinó que además, un poco más rápida. De totas formas, el hecho de que este poco sea una simple constante multiplicativa implica que nos mantenemos dentro de las mismas clases de eficiencia.

Encadenamiento separado

Una evolución lógica del hash por direccionamiento abierto surge cuando los lenguajes de programación permiten al programador su propia gestión de la memoria. Es decir, el encadenamiento separado surge con el uso de la memoria dinámica. Esencialmente, se trata de implementar los diccionarios en un vector de listas encadenadas.

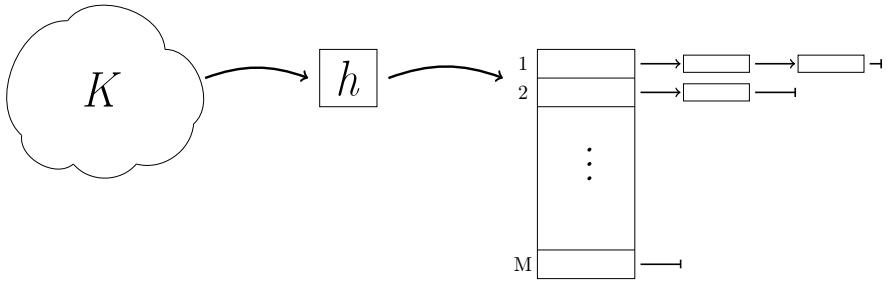


Figura 2.4: *Encadenamiento separado.*

En la Figura 2.4 se muestra una imagen gráfica de la implementación de las tablas de dispersión con encadenamiento separado. Se puede ver que de media, las cadenas tendrán longitud igual al factor de carga, n/M. Esta nueva visión nos soluciona dos de los inconvenientes de la implementación anterior:

- Ya no tendremos problemas de que el diccionario esté lleno. O almenos, tardará mucho más en llenarse y siempre se podrá posponer este momento haciendo ampliaciones de la memoria del ordenador. En definitiva, el tamaño del diccionario ya no será un cuello de botella.
- Las claves siempre irán al lugar que les corresponda según la función de dispersión. El hecho de que antes cualquier clave pudiera ir a parar a cualquier posición del vector introduce un cierto descontrol que en ningún caso resulta favorable.

El código que implementa una tabla de dispersión con la técnica de encadenamiento separado para el tratamiento de colisiones se puede ver en el Algoritmo 2.6.

```
class diccionario_hashing_separate_chaining {
    int n;
    nodo* T[1+M];
    int h(int k) {return 1 + k % M;}
public:
    diccionario_hashing_separate_chaining() {
        memset(T,NULL,(1+M)*sizeof(nodo*)); n=0;
    }
    void insertar(elemento e) {
        int i = h(e.clave);
        if (T[i] == NULL) {
            T[i] = new nodo;
            T[i]→e = e;
            T[i]→siguiente = NULL;
        }
        else {
            nodo* actual = T[i];
            while (actual→siguiente != NULL) {
                  actual = actual→siguiente;
            }
            actual→siguiente = new nodo;
            actual = actual→siguiente;
            actual→e = e;
            actual→siguiente = NULL;
        }
        ++n;
    }

    bool buscar(elemento& e) {
        int i = h(e.clave);
        nodo* actual = T[i];
        while (actual && actual→e.clave != e.clave)
            actual = actual→siguiente;
        if (actual != NULL) {
            e = actual→e;
            return true;
        }
        return false;
    }
};
```

Algoritmo 2.6 *Implementación de un diccionario en una tabla de dispersión con encadenamiento separado.*

Haciendo un breve análisis de eficiencia, se ve fácilmente que aparte de la creación que es $\Theta(1)$, las otras tres operaciones son todas $\Theta(n/M) = \Theta(n)$.

Así pues, a pesar de la evolución del hashing con encadenamiento separado respeto al de direccionamiento abierto, no ofrece ventajas cuantitativas respeto a la eficiencia, sí que mejora aquella implementación en el sentido de capacidad y de modularidad. Esta nueva implementación es una estructura más ordenada y de capacidad relativa al sistema donde sea desarrollada.

Fijaos sin embargo, que a pesar de que del capítulo anterior ya sabemos que una búsqueda en un espacio ordenado de acceso directo se resuelve en $\Theta(log(n))$, aún no hemos aprovechado este hecho en ninguna de las estructuras que tienen por finalidad precisamente eso, la búsqueda.

Por tanto, vamos por fin a diseñar estructuras más ágiles para esta operación.

2.2.3 Árboles Binarios de Búsqueda (BSTs)

Como es lógico, una vez disponibles las instrucciones que nos permiten gestionar la memoria desde el programa usuario (memoria dinámica), profundizamos en la explotación de este recurso en el momento de crear nuevas estructuras de datos. Los árboles binarios de búsqueda (binary search trees, en inglés) son estructuras profundamente dinámicas que cuando están vacías ocupan casi tan poco como un entero.

Se llama *árbol* a una estructura compuesta por nodos interrelacionados, en la que todos ellos, excepto uno de especial denominado *raíz* del árbol, tienen asociado un nodo, el *padre*. Si cada nodo tiene un padre distinto, entonces el árbol es una lista, y su primer elemento, la raíz.

Se califica de *binarios* a los árboles en los que como mucho dos nodos comparten un mismo padre. Estos dos nodos se llaman *hijo izquierdo* e *hijo derecho* del nodo que es su padre. Los nodos que no tienen ningún hijo se llaman *hojas*. Y a todos los otros nodos, que no son ni la raíz ni las hojas, se les llama nodos *intermedios*.

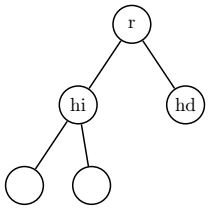


Figura 2.5: *Árbol binario.*

Gráficamente, todo esto se corresponde con la estructura arborescente mostrada en la Figura 2.5, donde se presenta un árbol con la raíz (r), su hijo izquierdo (hi) que es un nodo intermedio, y su hijo derecho (hd) que es una hoja. También se ve que del hijo izquierdo cuelgan dos hojas más que son sus hijos.

Utilizaremos los árboles binarios de búsqueda para implementar diccionarios. En cada nodo guardaremos un elemento, pareja <clave,descripcion>, como los mostrados en el Algoritmo 2.1 de la página 60.

Ha quedado claro, pues, por qué se les llama árboles binarios. Lo que los caracteriza como *de búsqueda*, además, es la invariante que siempre se debe cumplir en esta implementación de los diccionarios. Esta invariante se ha de satisfacer para cualquier subárbol, es decir, para toda la descendencia de cada nodo.

Definición 2.4 Invariante de los árboles binarios de búsqueda *La clave del elemento padre es mayor que la del elemento hijo izquierdo y menor que la del elemento hijo derecho.*

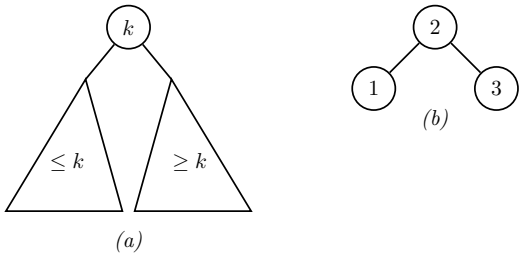


Figura 2.6: *(a) Invariante de los árboles binarios de búsqueda; (b) Ejemplo.*

Se puede ver la idea de la invariante en la Figura 2.6 *(a)*, y un ejemplo sencillo en la Figura 2.6 *(b)*.

De esta invariante de las claves de los diccionarios resulta una consecuencia immediata muy interesante de retener:

El recorrido en inorden[1] de un árbol binario de búsqueda recorre todas las claves por orden ascendente.

Implementación de los Árboles Binarios de Búsqueda

A partir de este punto nos adentramos en el uso de estructuras profundamente dinámicas, con sus constructores y sus destructores. Dispuestos a hacer un uso

[1] El recorrido inorden de un árbol empieza por el hijo izquierdo, sigue con la raíz, y luego por el hijo derecho. Todo recursivamente, es decir, cada paso, mientras se pueda.

intensivo de la memoria dinámica, el destructor adquiere una importancia que antes no tenía. Es necesario darse cuenta de las consecuencias que puede tener implementar una destrucción incorrecta de este tipo de estructuras. Consecuencias que a la larga conllevan la caída del sistema.

Probablemente una de las pifias más comunes de muchos programas comerciales es la mala implementación de los destructores. Es como dejarse la luz encendida cuando se sale de casa. Nadie se da cuenta. Todo parece que marche bien, y hasta final de mes cuando llega la factura nadie se para pensar si está haciendo las cosas como se debe. Esto mismo pasa con estas aplicaciones de poca calidad. Como muy difícilmente el usuario atribuirá los problemas que ocasiona una mala destrucción de las estructuras a la aplicación que las realiza, hay muchos programadores, e incluso muchas empresas de desarrollo de software que no le prestan la importancia que realmente tiene. De hecho, la única manera de percibir que una aplicación no libera el espacio que ocupa es monitorizando la cantidad de espacio que utiliza y abrir y cerrar la aplicación varias veces para comprobar si la memoria usada va incrementando. Y aún así, tampoco podemos estar seguros.

Aquí se trata de aprender a codificar aplicaciones de calidad. Por tanto, consideraremos muy seriamente los destructores siempre que hablemos de creación de estructuras de datos. Está claro que si una estructura está mal creada, el programa que la pretende utilizar peta, y el usuario no puede hacer lo que desea. Por eso, los constructores siempre funcionan bien. En cambio, cuando los destructores funcionan malamente, la única pista que tenemos es que después de utilizar una aplicación varias veces, cae el sistema. Además, probablemente, la caída se produzca en un momento en que ni siquiera estemos utilizando la aplicación responsable de esta mala gestión.

Así pues, que quede claro. Cuando tenemos una clase que se llama *tomate*, de la misma manera que el código de creación de las instancias (el constructor) se debe llamar *tomate()*, el destructor se llamará $\sim$*tomate()*. Y aunque no lo notemos como usuarios de la aplicación, como desarrolladores sí que lo vemos bien claro ya que el entorno que utilizamos para depurar nos lo informa con insistencia por medio de las fugas de memoria, o en inglés, los *memory leaks.*

Bien, como se puede observar en el Algoritmo 2.7, la definición de árbol binario de búsqueda, igual que con las listas enlazadas, queda reducida casi a un apuntador a un nodo. Estas implementaciones son inmejorables en cuanto a eficiencia espacial. Un árbol vacío ocupa tan poco como puede, que significa como un entero más o menos. Y nunca excederá $\Theta(n)$, siendo n el número de elementos almacenados en el diccionario. Las tablas de símbolos no tenían esta cualidad.

En el Algoritmo 2.7 se muestra una posible declaración para una clase que implementa un diccionario en un árbol binario de búsqueda. La estructura *nodo* del árbol se mantiene privada reforzando la modularidad. Por otro lado, la función de inserción no inserta si la clave ya estaba en el diccionario.

```
class diccionario_arbol_binario_de_busqueda {
    struct nodo {
        elemento e;
        nodo* hi;
        nodo* hd;
    }
    nodo* raiz;
    int n;

    void liberar(nodo* r) {
        if (r) { liberar(r→hi); liberar(r→hd); delete r;}
    }

    bool _insertar(nodo*& r, elemento e) {
        if (r == NULL) {
            r = new nodo;
            r→e = e;
            r→hi = NULL;
            r→hd = NULL;
            n++;
            return true;
        }
        if (e.clave < r→e.clave) return _insertar(r→hi,e);
        if (e.clave > r→e.clave) return _insertar(r→hd,e);
        if (e.clave == r→e.clave) return false;
        return true;
    }

    bool _buscar(nodo* r, elemento& e) {
        if (r == NULL) return false;
        if (r→e.clave == e.clave) {
            e = r→e;
            return true;
        }
        if (e.clave < r→e.clave) return _buscar(r→hi,e);
        if (e.clave > r→e.clave) return _buscar(r→hd,e);
        return true; // para el compilador
    }
public:
    diccionario_arbol_binario_de_busqueda() {raiz = NULL; n=0;}
    ~diccionario_arbol_binario_de_busqueda() {liberar(raiz);}
    bool insertar(elemento e) {return _insertar(raiz,e);}
    bool buscar(elemento& e) {return _buscar(raiz,e);}
};
```

Algoritmo 2.7 *Declaración de la clase para la implementación de un diccionario en un árbol binario de búsqueda.*

Como variables miembro de la clase, todas ellas privadas también, tenemos la raíz, que es un simple apuntador a un nodo, y el número n de elementos presentes en el diccionario, que no es imprescindible, pero es muy útil. También dentro de la parte privada se implementan las operaciones internas. Para los casos en que las operaciones públicas se implementan tan sólo como un envoltorio de alguna operación interna (insertar y buscar), la interna comienza con el prefijo "_".

Ya en el ámbito de visibilidad pública tenemos en primer lugar la declaración del constructor y el destructor. Seguidamente se presentan las operaciones que caracterizan una estructura de datos como diccionario, implementándolo con árboles binarios de búsqueda. Antes de acabar la sección, también se estudiarán las operaciones de eliminación y de recorrido.

A lo largo de las implementaciones que siguen supondremos que no hay valores de clave repetidos. En la Figura 2.7 se puede ver un ejemplo del cual haremos referencia en las próximas secciones. Lo que se muestra en el interior de los nodos es el valor de las claves de los elementos que almacenan.

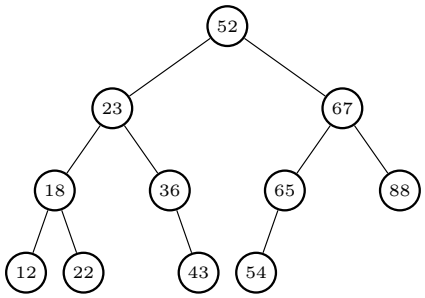


Figura 2.7: *Ejemplo utilizado en la implementación de las operaciones.*

Construcción y Destrucción

Tal como se ha visto la clase *diccionario_arbol_binario_de_busqueda* del Algoritmo 2.7 consta fundamentalmente de un apuntador a un nodo. Así pues, crear un diccionario significa tan sólo inicializar el apuntador a NULL, y poner a cero el número de elementos, tal como se puede ver en el Algoritmo 2.8.

Esta bien claro que esta implementación de la operación de creación es $\Theta(1)$.

Una vez creado, el árbol binario de búsqueda está preparado para almacenar los elementos que se le quieran insertar. Para esto se requiere que estos elementos sean de la forma vista en la estructura del Algoritmo 2.1, de la página 60. Es decir, que estén formados por una clave y una descripción o información asociada.

El destructor no es tan sencillo. Como se puede ver en el Algoritmo 2.8

hace uso de una función recursiva llamada *liberar* que recibe como parámetro la raíz del subárbol que se libera. Esta función es privada porque en ningún caso conviene que el usuario (programador que utiliza esta estructura) tenga acceso a la estructura dinámica que soporta el árbol. Es una función sencilla que tiene por caso trivial cuando el parámetro sea NULL, es decir, que sea algún hijo de una hoja. O sea, empieza liberando el hijo izquierdo, después el hijo derecho, y finalmente, la raíz. No se reqiuere ningún orden para las dos primeras llamadas, pero sí que en cambio hay que borrar en última instancia la raíz por razones suficientemente claras.

```
private:
    void liberar(nodo* r) {
        if (r != NULL) {
            liberar(r→hi);
            liberar(r→hd);
            delete r;
        }
    }

public:
// constructor
    diccionario_arbol_binario_de_busqueda() {
        n = 0;
        raíz = NULL;
    }

// destructor
    ∼diccionario_arbol_binario_de_busqueda() {
        liberar(raíz);
    }
```

Algoritmo 2.8 *Creación y destrucción de un árbol binario de búsqueda.*

A modo de gimnasia intelectual, observad que para destruir el árbol del ejemplo de la Figura 2.7 la secuencia de nodos que se borraría sería 12, 22, 18, 43, 36, 23, 54, 65, 88, 67 y 52.

La eficiencia de la destrucción se puede ver por el número de nodos que trata. Será $\Theta(n)$. Calcularla utilizando los Teoremas Maestros podría resultar no tan sencillo, como se verá cuando se hable de las eficiencias en general de todas las operaciones de los árboles binarios de búsqueda.

Inserción

Exclusivamente insertaremos hojas. Este hecho es importante e impactará en la eficiencia. No es deseable en absoluto, pero ésta es la manera como lo sabemos hacer. En definitiva, el primer elemento que se inserte en un árbol binario de búsqueda será la raíz. Y, por muchas inserciones que se hagan después, nunca más habrá un cambio en este aspecto. El primer nodo ocupará el lugar de la raíz y no tenemos ningún método para cambiarla. La única cosa que podemos hacer si queremos cambiar la raíz es eliminarla y volver a insertar el elemento. Entonces, igualmente, la nueva raíz se quedará para siempre.

Establecemos el convenio de asignar el valor NULL para los hijos de las hojas del árbol, cosa que ya se ha utilizado en el caso trivial del destructor. Hay que realizar estas asignaciones en el momento de insertar nuevos elementos.

Observad que un árbol vacío mantiene la invariante de los árboles binarios de búsqueda. Impondremos esta propiedad en la inserción para poder demostrar que en todo momento se mantendrá. Haciéndolo de esta forma, la demostración podría formalizarse por inducción sobre el tamaño del árbol, n.

En el Algoritmo 2.9 se puede observar el código que implementa la operación privada *_insertar()*. En la primera línea se mira si el subárbol que cuelga del apuntador r es vacío. Si es así, significa que ya hemos encontrado el punto de inserción. Entonces tan sólo hay que guardarse el elemento a insertar como el elemento del nodo recién creado. Como todas las inserciones se hacen como hojas del árbol, siempre que se inserte un nuevo elemento, se asigna el valor NULL a sus dos hijos.

```
bool _insertar(nodo*& r, elemento e) {
    if (r == NULL) {
        r = new nodo;
        if (r == NULL) return false;
        r→e = e;
        r→hi = NULL;
        r→hd = NULL;
        ++n;
        return true;
    }
    if (e.clave < r→e.clave) return _insertar(r→hi,x);
    if (e.clave > r→e.clave) return _insertar(r→hd,x);
    if (e.clave == r→e.clave) return false;
    return true;
}
```

Algoritmo 2.9 *Inserción en un árbol binario de búsqueda.*

Para el caso que r no sea NULL, deberemos ir cayendo por la rama del árbol correspondiente a la nueva clave. Este posicionamento se hace con las dos llamadas recursivas de la parte final del código de la función.

El parámetro r, la raíz del subárbol donde se debe hacer la inserción, se pasa por referencia ya que en el cuerpo de la función puede tenerse que modificar. En cambio, el elemento e en ningún caso se modificará y por eso puede ser pasado por valor.

Si en el árbol del ejemplo de la Figura 2.7 insertáramos un elemento con clave 34, iría a rellenar el espacio correspondiente al hijo izquierdo del nodo 36, tal como se puede ver en la Figura 2.6.

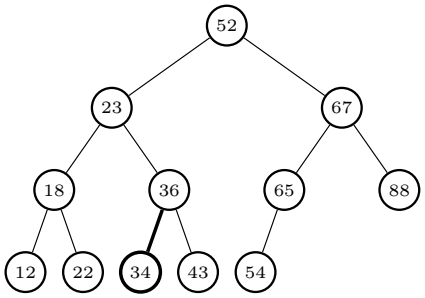


Figura 2.8: *Inserción de un nuevo nodo con clave* 34 *en el árbol de la Figura 2.7.*

Búsqueda

Como antes, no hay demasiada diferencia entre insertar y buscar, ya que la tarea más pesada de insertar es precisamente buscar el punto de inserción.

En el Algoritmo 2.10 se muestra la forma de hacer una búsqueda en el árbol. Fijaos que en este caso la función retorna un valor booleano que indicará si se ha encontrado el elemento o no. Además, el elemento a buscar se recibe como parámetro por referencia. Así pues, para buscar un elemento en el árbol, el módulo que llame a esta función deberá rellenar el valor de la clave del elemento e que desee buscar. Si efectivamente se encuentra el elemento se retornará el valor booleano *cierto* y se rellenará *e.descripcion* con la información asociada a este elemento. En caso que el elemento no esté en el árbol, se retornará *falso* y no se tocará el contenido del elemento e.

```
bool _buscar(nodo* r, elemento& e) {
    if (r == NULL) return false;
    if (e.clave == r→clave) {
        e = r→e;
        return true;
    }
    if (e.clave < r→clave) return _buscar(r→hi,e);
    if (e.clave > r→clave) return _buscar(r→hd,e);
    return false;
}
```

Algoritmo 2.10 *Búsqueda en un árbol binario de búsqueda.*

Hagamos una ojeada a la eficiencia de las dos operaciones vistas hasta ahora. Para poder decir alguna cosa, observemos primero de todo que la eficiencia depende fuertemente de la disposición del árbol. Y antes de profundizar en el tema, son necesarias un par de definiciones.

Definición 2.5 Altura de un nodo. *Cantidad de descendientes que tiene un nodo por el camino más largo hasta llegar a las hojas.*

Así pues, si llamamos $h(r)$ a la altura de un nodo (o del subárbol con raíz en r), tenemos la definición recursiva...

$$h(r) = \begin{cases} 0 & \textit{si } r \textit{ es una hoja.} \\ 1 + max\{h(hi(r)), h(hd(r))\} & \textit{en otros casos.} \end{cases}$$

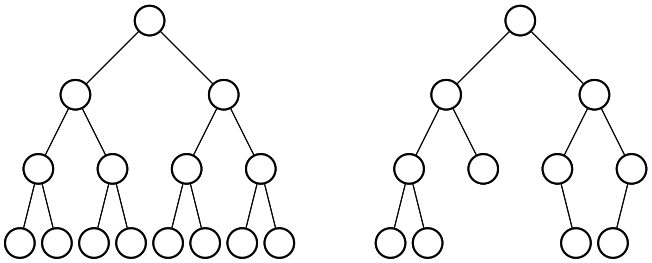
Las hojas tienen altura 0, y cualquier otro nodo tiene la altura igual a la máxima entre las de sus dos hijos, más 1. La definición de la altura de un nodo es en cierta manera la inversa a la de *nivel* de un nodo. Decimos que la raíz está en el nivel 0, sus hijos a nivel 1, y así sucesivamente. Cuando en un árbol todas las hojas se encuentran en el máximo nivel, es decir, a bajo del todo, entonces el árbol es *completo*. El concepto de altura de un nodo nos permite introducir un adjetivo más a los árboles binarios...

Definición 2.6 Árbol equilibrado *Aquel árbol que para cualquier nodo, la diferencia de alturas entre sus dos hijos es* 1 *como máximo.*

$$r \textit{ equilibrado} \Leftrightarrow |h(hi(r)) - h(hd(r))| \leq 1.$$

En rigor, no se debería dar la eficiencia de las operaciones en función de la altura h, ya que por definición hay que darla en función del tamaño de los datos

n, y no por su distribución ni ningúna otra propiedad cualitativa. De totas formas, a menudo se utiliza esta forma de expresar la eficiencia con la intención de hablar del caso medio.



(a) *(b)*

Figura 2.9: *(a) Árbol completo (y equilibrado); (b) Árbol equilibrado.*

De la definición de árbol equilibrado se desprende que los árboles completos son equilibrados. En ese caso, $h \backsimeq log(n)$, siendo este logaritmo en base 2 porque estamos hablando de árboles binarios. En la Figura 2.9*(a)* se puede ver un árbol completo, y por tanto equilibrado. En la Figura 2.9*(b)* se muestra un árbol equilibrado cualquiera. Ejemplos de árboles desequilibrados se pueden observar en la Figura 2.10.

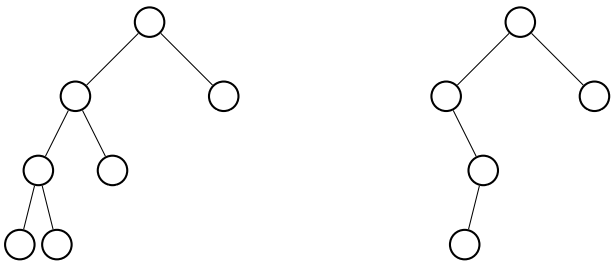


Figura 2.10: *Árboles desequilibrados.*

Una vez definido el concepto de árbol equilibrado, hablemos de eficiencia. Lo que se dice seguidamente vale tanto para la operación de insertar como para la de buscar:

caso mejor: El árbol es completo, entonces en cada llamada recursiva reducimos el tamaño del subproblema a la mitad. Para estos casos, según el Teorema Maestro II, tenemos que $a = 1$, $b = 2$, y $k = 0$. Por tanto, las eficiencias de las dos operaciones son $\Theta(log(n))$.

caso peor: El árbol no sólo resulta desequilibrado sinó que adquiere forma de lista, como se ve en la Figura 4.6. En estos casos se dice que el árbol degenera, $h \backsimeq n - 1$, y la reducción del tamaño del subproblema ya no se puede

considerar divisora, sinó substractora. Entonces vamos a parar al Teorema Maestro I, con $a = 1$, $c = 1$, y $k = 0$. Les eficiencias en este caso peor serán $\Theta(n)$.

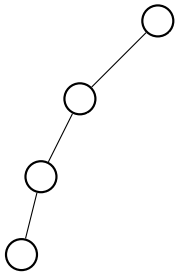


Figura 2.11: *Árbol degenerado.*

Siempre podemos decir que las operaciones son $\Theta(h)$, ya que para los árboles equilibrados $h \backsimeq log(n)$ mientras que para los degenerados $h \backsimeq n - 1$. De totas formas, como se ha dicho, hablar de h no es hablar del tamaño de los datos, sinó del contenido.

Eliminación

La eliminación es más complicada que las dos operaciones anteriores. Es importante mantener la invariante después de una eliminación. Para conseguirlo, hay que distinguir diferentes casos según el número de hijos del nodo que se quiera eliminar. Comenzamos por el caso más fácil. Cuando se trata de eliminar una hoja podemos hacerlo sin ningún problema. Sencillamente, liberamos el espacio que ocupa y ponemos el apuntador de su padre a NULL.

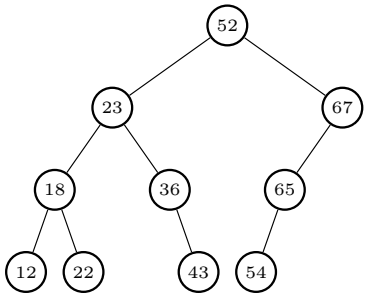


Figura 2.12: *Estado después de eliminar el nodo con clave* 88 *del árbol de la Figura 2.7.*

En la Figura 2.12 se puede ver el resultado de la eliminación de una hoja del árbol del ejemplo de la Figura 2.7.

Lo que se muestra en el interior de los nodos de las Figuras 2.12, 2.13, y 2.14 también representa el valor de sus claves. Observad que tanto antes, en la Figura 2.7, como después de la eliminación, Figura 2.12, se mantiene la invariante.

Para el caso de tener que eliminar un nodo que sólo tiene un solo hijo, la operación también es sencilla. Ponemos el hijo en el lugar del nodo que queramos eliminar.

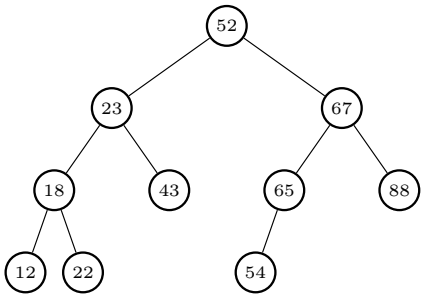


Figura 2.13: *Estado después de eliminar el nodo con clave* 36 *del árbol de la Figura 2.7.*

Y finalmente, para el caso de tener que eliminar un nodo de dos hijos, la cosa se complica.

Unas reflexiones previas nos ayudarán a atacar el problema:

- En una estructura de datos vector, cuando eliminamos un elemento, acostumbramos a desplazar todos los siguientes una posición anterior. Los vectores tienen un inicio que siempre es la primera posición.
- En un árbol binario de búsqueda, el valor de clave inmediatamente inferior a un nodo de dos hijos es el máximo de los valores que cuelgan del subárbol de su hijo izquierdo, que existe. Y el inmediatamente superior, el mínimo de su hijo derecho, que también existe.

Para eliminar en un árbol binario de búsqueda utilizaremos una filosofía parecida a la de los vectores. Desplazaremos los elementos posteriores al que queremos borrar a una posición anterior a la que se encuentran. A diferencia de los vectores, sin embargo, un árbol binario de búsqueda no tiene una primera posición fija, sinó que es flotante. Eso nos evitará realizar el desplazamiento de cada uno de los elementos. Podemos escoger entre dos operaciones equivalentes:

- Podemos colocar el nodo con el valor inferior de clave en la posición del que queremos eliminar.
- Podemos colocar el nodo con el siguiente valor de clave en la posición del que queremos eliminar.

Una opción razonable es aprovechar este grado de libertad para mantener el árbol tan equilibrado como sea posible.

Además, podemos enunciar la siguiente

Proposición 2.1 *Si en un árbol binario de búsqueda un nodo tiene dos hijos, el nodo con la clave immediatamente superior no puede tenerlos.*

Prueba *Sea k el valor de la clave de un nodo con los dos hijos no vacíos. El nodo correspondiente a la clave con valor inmediatamente superior, k', debe colgar de su hijo derecho, que existe por hipótesis. Si este nodo tuviera dos hijos no vacíos, su hijo izquierdo tendría una clave k'' superior a k e inferior a k' por definición de la invariante de los árboles binarios de búsqueda. Esto es una contradicción ya que k' ha sido definida como el valor de clave inmediatamente superior a k.* □

El hecho que tanto el inferior como el posterior de un nodo de dos hijos deban tener un solo hijo o bien sean hojas nos garantiza que para borrar un nodo de dos hijos podemos borrar el anterior, o el posterior, que no serán de dos hijos, guardando sus valores de clave e información para reubicarlo donde estaba el que queremos eliminar.

A la operación de sacar un nodo del árbol sin borrarlo, o sea, sencillamente desatarlo, le llamamos *sacar* el nodo.

Para el caso, utilizaremos el primer criterio. Eliminar un nodo de dos hijos constará de dos pasos. Primero, sacaremos del árbol el que tenga la clave inmediatamente inferior y luego lo colocaremos en el lugar del nodo que se quiere borrar.

Así pues, para eliminar un nodo de dos hijos conviene implementar primero una función para sacar el nodo anterior. En el Algoritmo 2.11 se muestra una posible implementación de la función que nos interesa. No hace más que sacar el máximo de un árbol, que no tendrá hijo derecho, retornando un apuntador al nodo sacado. O sea, no se hace ningún *delete.*

```
nodo* sacar_maximo(nodo*& r) {
    if (r→hd) return sacar_maximo(r→hd);
    nodo* s = r;
    r = r→hi;
    return s;
}
```

Algoritmo 2.11 *El nodo con clave máxima de un árbol binario de búsqueda no vacío se saca del árbol y se retorna el apuntador que lo señala.*

1a. línea: Nos colocamos, recursivamente, sobre el nodo con valor de clave máximo.

2a. línea: Nos guardamos un apuntador a este nodo para poderlo retornar una vez desatado.

3a. línea: Desatamos el nodo del árbol sin borrarlo, ya que su hermano (hijo derecho de su padre) pasa a ser su padre.

4a. línea: Retornamos el apuntador apuntando al nodo que hasta hace un momento era el de máxima clave.

Es una función bien sencilla, perteneciente a $\Theta(h)$.

Una vez disponible la función implementada en el Algoritmo 2.11, se puede resolver la operación de eliminación distinguiendo entre casos. En el Algoritmo 2.12 se muestra una implementación de la operación de eliminar. Como antes, en este algoritmo la operación *_ eliminar()* es privada. Entonces se ofrece un envoltorio en la visión pública, para al código usuario de la estructura. Se retorna un booleano diciendo si se ha borrado el elemento, o por el contrario, no estaba presente en el árbol.

El Algoritmo 2.12 se explica seguidamente línea a línea...

```
private:
bool _eliminar(nodo*& r, elemento e) {
    if (r == NULL) return false;
    if (e.clave < r→e.clave) return eliminar(r→hi,e);
    else if (e.clave > r→e.clave) return eliminar(r→hd,e);
    else {
        nodo* s = r;
        if (r→hi == NULL) r = r→hd;
        else if (r→hd == NULL) r = r→hi;
        else{
            nodo* m = sacar_maximo(r→hi);
            m→hd = r→hd; m→hi = r→hi; r = m;
        }
        delete s; --n;
    }
    return true;
}
public:
bool eliminar(elemento e) {return _eliminar(raíz,e);}
```

Algoritmo 2.12 *Eliminación en un árbol binario de búsqueda.*

1a. línea: Cabecera de la función. Se recibe el árbol por referencia ya que será modificado. También se recibe el elemento aunque sólo se utilizará su clave.

2a. línea: Se comprueba que no se haya llegado a las hojas sin haberse encontrado el nodo a eliminar. Si así fuera, retornaríamos falso.

3a. línea: Posicionamiento recursivo a la izquierda sobre el nodo a borrar.

4a. línea: Posicionamiento recursivo a la derecha sobre el nodo a borrar.

5a. línea: Distinción de casos. Observad que una vez llegados a este punto ya seguro que acabamos haciendo el delete y reduciendo el número de elementos.

6a. línea: Guardamos el apuntador al nodo que finalmente borraremos.

7a. línea: Si el nodo a borrar no tiene hijo izquierdo, lo cual incluye el caso de nodos hojas, lo quitamos del árbol, poniendo a su hijo derecho en su lugar (que en caso de nodos hoja valdrá NULL).

8a. línea: Sólo para los nodos que tengan exclusivamente hijo izquierdo. Lo quitamos del árbol, poniendo a su hijo izquierdo en su lugar.

9a. línea: Para los nodos de dos hijos. Sacamos del árbol el nodo con clave inmediatamente inferior (es decir, la máxima de las claves inferiores) manteniéndolo en un apuntador nuevo.

10a. línea: Atamos por los tres lados el nodo sacado. Lo dejamos en la posición de la raíz.

11a. línea: Liberamos la memoria que ocupaba el nodo borrado, y reducimos el número de elementos.

En la Figura 2.14 se muestra el resultado de eliminar el nodo con clave 23 del árbol de la Figura 2.7.

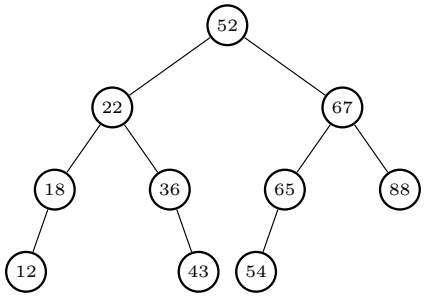


Figura 2.14: *Estado después de eliminar el nodo con clave* 23 *del árbol de la Figura 2.7.*

Respeto a la eficiencia del algoritmo de eliminación podemos decir exactamente lo mismo que del de inserción o de búsqueda. Esto es, $\Theta(h)$. De todas

formas, como ya se ha dicho antes, decir que este algoritmo es $\Theta(h)$ no es responder a la eficiencia, ya que h no es el tamaño de los datos. Y de hecho, está más relacionado con el contenido que con el tamaño. La única cosa que podemos decir con certeza, es que en el caso mejor es $\Theta(log(n))$ y en el peor $\Theta(n)$.

Recorrido

La función para recorrer un árbol binario de búsqueda por orden ascendente de clave no hace más que el recorrido inorden de un árbol binario. Se puede observar en el Algoritmo 2.13 la correspondiente versión privada. Por el hecho de ser un árbol no hay ciclos, y por tanto, pertenece a $\Theta(n)$.

```
void _mostrar(nodo* r) {
    if (r) {
        mostrar(r→hi);
        tratar(r);
        mostrar(r→hd);
    }
}
```

Algoritmo 2.13 *Recorrido ascendente de un árbol binario de búsqueda.*

La función pública para mostrar sería un envoltorio de ésta. No recibiría ningún parámetro, y llamaría la del Algoritmo 2.13 con la variable miembro *raíz*.

Con los BSTs nos acercamos considerablemente al tipo de estructura que andamos buscando. Por la naturaleza de la búsqueda dicotómica, vista en el capítulo anterior, algo nos dice que las eficiencias logarítmicas en los casos medio y mejor deben ser inmejorables. Hemos topado con la estructura que estábamos buscando y que de todo aquello visto hasta ahora, es de largo la mejor implementación para los diccionarios, ya que la mejora no se mide en constantes ocultas, sinó en clases de eficiencia.

Aún así, probemos de rematar la tarea mejorando los tiempos de las operaciones para los casos peores. Seguidamente se presenta la última implementación para los diccionarios, que muy dudosamente puede ser mejorada.

2.2.4 Árboles AVL


G.M. Adelsón-Velski

Georgi Maksímovich Adelson-Velski (1922-...) es, y Yevgeni Landis (1921-1997) fue, matemáticos. El año 1962 idearon el árbol autoequilibrado, la variante de los árboles binarios de búsqueda que se explica en esta sección. De sus iniciales, el nombre AVL. De Adelson se dice que el año 1965 lideró un proyecto en el que se desarrollaba uno de los primeros programas para jugar al ajedrez de la historia de la computación, en el Instituto de Física Teórica y Experimental de Moscú.

Introducción

Cuando se ha definido la eficiencia en el capítulo anterior se ha decidido ignorar el orden en que llegan los datos a los algoritmos. Hemos concluido que la eficiencia dependería exclusivamente del tamaño, ignorando el hecho de que para instancias de un mismo tamaño, un mismo algoritmo puede tardar tiempos muy diferentes en dar respuesta. Ciertamente, se ha dicho que si para distintos ordenaciones de los datos los tiempos varían mucho, entonces distinguiríamos entre casos mejor, medio, y peor. Y también se ha visto la repercusión que tienen las sentencias alternativas en la distinción de estos casos.

Por otro lado, el código con el cual se han implementado los árboles binarios de búsqueda, en la Sección 2.2.3, utiliza intensivamente sentencias alternativas que gobiernan el flujo de ejecución por medio de las llamadas recursivas. También se puede comprobar fácilmente que tal como se ha implementado la inserción, una vez el primer nodo ha sido insertado en un árbol, ya jamás dejará de ocupar la raíz (a no ser que sea eliminado).

Es decir, tal como se construyen los árboles binarios de búsqueda, el orden en que llegan los datos queda latente.

Con todo, concluimos que la implementación de las operaciones para los árboles binarios de búsqueda están en las antípodas de los algoritmos con valores predictibles de eficiencia. Por esta razón, en todo momento ha sido necesario distinguir entre caso mejor y caso peor, y en ningún caso, ni tan sólo hemos sido capaces de establecer un caso medio. Todo junto, depende en exceso del orden en que llegan los datos.

Como se ha dicho en la Sección 2.2.3, un árbol es desequilibrado si existe algún nodo con una diferencia entre alturas de sus dos hijos mayor que 1. La altura de una hoja es cero. Un árbol degenerado es lo máximo de desequilibrado posible, es decir, una lista.

Resolver la degeneración

Con los árboles binarios de búsqueda, por un lado tenemos el problema de la degeneración, que arrastra las eficiencias hacia $\Theta(n)$. Por otro lado, de su definición, tenemos un margen de libertad. Está claro que los dos árboles de la Figura 2.15 son equivalentes para almacenar un diccionario.

Figura 2.15: *Árboles binarios de búsqueda equivalentes.*

La idea, pues, es que si tenemos un árbol como el de la Figura 2.15, nos lo coloquemos como más nos interese ante nuevas inserciones. Eso significa que si llega un nuevo elemento con valor de clave $k > 20$, entonces representaremos el árbol como el de la Figura 2.15*(a)*. En cambio, si el nuevo nodo tuviera por clave $k < 10$, entonces utilizaríamos la versión de la Figura 2.15*(b)*. Además, también convendría ser capaces de poder insertar un nuevo elemento con clave $k = 15$ como nueva raíz del árbol. O sea, se trataría de insertar buscando siempre el equilibrio.

A medida que el árbol va creciendo, el análisis de la representación más conveniente según la nueva entrada puede resultar cada vez más difícil. La gracia de los AVLs es que después de cada inserción, determinan de manera local a la rama donde se cuelga el nodo insertado, cuál es la operación de balanceo necesaria.

Implementación de los AVLs

Referente al espacio de memoria, lo único addicional que utilizaremos para los árboles AVL será un entero donde guardaremos la altura de los nodos. Eso es, del Algoritmo 2.7 de la página 79, donde se presentaba una implementación completa para los árboles binarios de búsqueda, redefinimos la estructura de los nodos quedando como se muestra en el Algoritmo 2.14.

```
struct nodo {
    elemento e;
    nodo* hi;
    nodo* hd;
    int h;
};
```

Algoritmo 2.14 *Estructura para los nodos de un árbol* AVL.

Para mantener el árbol equilibrado en inserciones y eliminaciones habrá que gestionar correctamente este nuevo miembro de la estructura *nodo*. En el Algoritmo 2.15 se puede observar la implementación de una función sencilla para poder saber la altura de un nodo en cualquier momento. Como parámetro de entrada, esta función recibe un apuntador al nodo. Así, se puede retornar un -1 cuando el apuntador sea NULL, es decir, para los hijos de las hojas. Con la función del Algoritmo 2.15, no sólo se obtiene la altura del nodo apuntado por r, sinó que además, se guarda cada vez que se calcula.

```
int calcular_altura(nodo* r) {
    if (r == NULL) return -1;
    r→h = 1 + max(calcular_altura(r→hi),
                  calcular_altura(r→hd));
    return r→h;
}
```

Algoritmo 2.15 *Cálculo y actualización de la altura del nodo apuntado por* r.

Está claro que sólo será necesario llamar a la función del Algoritmo 2.15 cuando la altura del nodo r pueda haber variado. Esto ocurrirá cada vez que se produzca una inserción, o una eliminación, en alguno de los dos subárboles que cuelguen de r. En cualquier otro momento, la altura estará actualizada en la misma variable miembro h.

Con el tratamiento de las alturas implementado, podemos proveernos de algunas operaciones que tendrán por finalidad equilibrar el árbol. Primero se verán las operaciones internas correspondientes a las inserciones. Después, con un par de operaciones internas adicionales, se podrá implementar la eliminación.

Los árboles AVL funcionan corrigiendo el más pequeño error de desequilibrio que se pueda producir. No veremos ningún procedimento útil para reequilibrar un árbol desequilibrado. No es así. Lo que hacen los AVLs es no permitir nunca que ningún nodo del árbol se desequilibre. Desde el momento de la

creación cuando todavía no hay elementos, jamás ocurrirá que un nodo tenga una diferencia de alturas entre hijos superior a 1. Tampoco nunca sucederá que simultáneamente haya dos nodos desequilibrados. Es una filosofía de resolución de conflictos, es decir, de ir apagando fuegos tras cada inserción y cada eliminación.

Operaciones privadas para la inserción en los árboles AVL

Vamos a entender bien las operaciones que pretendemos implementar.

En primer lugar, hay que observar los dos árboles desequilibrados más sencillos que existen, y las transformaciones necesarias para reequilibrarlos.

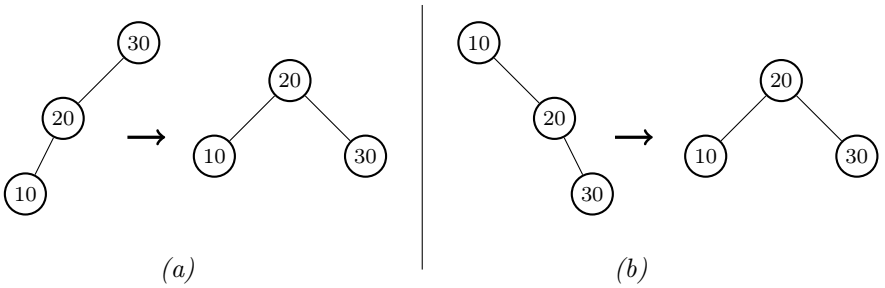


Figura 2.16: *(a) Desequilibrio izquierda_izquierda, y rotación simple que lo corrige; (b) Desequilibrio derecha_derecha y rotación simple que lo corrige.*

En la Figura 2.16 hay representadas las dos situaciones. En la parte izquierda, Figura 2.16*(a)*, se puede ver lo que se conoce como desequilibrio izquierda_izquierda. Se denomina así porque viene provocado por un hijo izquierdo de un hijo izquierdo. La operación para equilibrar tal situación se denomina rotación simple izquierda_izquierda. Con ella se obtendrá el árbol equivalente que hay en la parte derecha de la Figura 2.16*(a)*. Una definición análoga, tanto para el desequilibrio como para la operación reequilibradora, en este caso derecha_derecha, se muestra en la Figura 2.16*(b)*.

Bien, se trata de detectar las situaciones de desequilibrio mostradas en la Figura 2.16 y corregirlas justo después de que se produzcan. En los Algoritmos del 2.16 al 2.19 se implementan las correcciones. Después, en el Algoritmo 2.20 se muestra como habrá que modificar la inserción de los árboles binarios de búsqueda para mantener la estructura equilibrada.

```
void izquierda_izquierda(nodo*& r) {
    nodo* s = r;
    r = r→hi;
    s→hi = r→hd;
    r→hd = s;
    calcular_alcada(r);
    calcular_alcada(s);
}
```

Algoritmo 2.16 *Rotación simple izquierda_ izquierda sobre un nodo apuntado por r con hijo izquierdo no vacío.*

A continuación, se relata un seguimiento exhaustivo de la función del Algoritmo 2.16 con el ejemplo de la Figura 2.16*(a)*.

1a. línea: Se llama con el apuntador $nodo*\ r$ apuntando al nodo con clave $k = 30$. Eso es $r \rightarrow e.clave = 30$, pasando el parámetro por referencia. Es decir, la dirección de r, que viene a ser $\&r$.

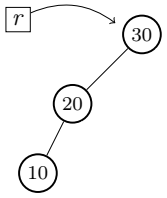


Figura 2.17: *1a. Línea*

2a. línea: Se declara, como variable local a la función, un nuevo apuntador s, donde se guarda el nodo raíz del subárbol con el cual se ha hecho la llamada, de momento apuntado por r. La dirección del apuntador s no es relevante. La dirección de r, en cambio, sí que lo es, ya que r está ubicado en el módulo que ha llamado esta función, y si se modificara r, se estaría cambiando el contenido remoto que nos apunta.

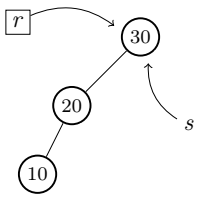


Figura 2.18: *2a. Línea*

3a. línea: Sin miedo de perder la memoria ocupada por el nodo raíz (ya que se está apuntando desde el apuntador s), modificamos r. A partir de ahora apunta al hijo izquierdo (con valor de clave inferior al de la raíz apuntada por s). Ahora, $s \rightarrow e.clave = 30$ y $r \rightarrow e.clave = 20$. Aunque hasta ahora no había aparecido el nodo (de hecho inexistente) hijo derecho de la clave 20, en la Figura 2.19 sí que aparece porque entra en acción.

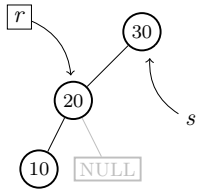


Figura 2.19: *3a. Línea*

4a. línea: El hijo derecho del nodo apuntado por r no existe en el ejemplo. Si existiese, tendría un valor de clave mayor que el de r y menor que el de s. En esta asignación, según el ejemplo, se asigna el valor NULL. No obstante, podría ser cualquier subárbol. En este paso se está haciendo un cambio de padre a todo este subárbol, de 20 a 30.

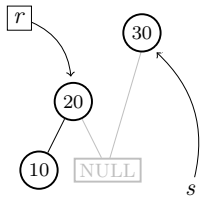


Figura 2.20: *4a. Línea*

5a. línea: El nodo apuntado por s puede ponerse como nuevo hijo derecho del apuntado por r, ya que el subárbol que hasta ahora estaba apuntado por el hijo derecho de r ya ha sido reparentizado y ya es hijo de s. Habiendo pasado r por referencia, cuando se sale, la nueva raíz es el nodo que cuando se entraba era el hijo izquierdo de la raíz de entrada.

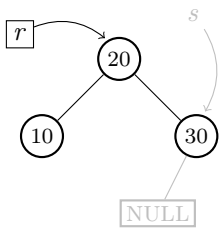


Figura 2.21: *5a. Línea*

En las Figuras 2.19, 2.20, y 2.21 los nodos con valor NULL aparecen en gris porque realmente no son ningún nodo. En todo momento hubieran podido estar dibujados así, pero tan sólo se han mostrado en el momento de intervenir en la acción.

En la Figura 2.21, el apuntador s aparece en gris porque es una variable local a la función, y por tanto dejará de existir en cuanto se retorne.

Finalmente, después de los movimientos de apuntadores, en la misma función se actualizan las alturas de los nodos ya que es en este punto donde han variado.

En total, se ha requerido un tiempo perteneciente a $\Theta(h)$. Ahora sí que ya es legítimo utilizar la altura como tamaño del árbol, porque en todo momento hablamos de árboles equilibrados, y por tanto, $h = log(n)$.

De manera análoga se podría proceder con un análisis para el caso de la rotación simple derecha_derecha. Esquivamos hacerlo porque sería del todo parecido al caso de la rotación simple izquierda_izquierda y resultaría fatigoso. Tan sólo se muestra, en el Algoritmo 2.17, una implementación de la función de rotación simple derecha_derecha a fin de observar la simetría entre las dos acciones.

```
void derecha_derecha(nodo*& r) {
    nodo* s = r;
    r = r→hd;
    s→hd = r→hi;
    r→hi = s;
    calcular_altura(r);
    calcular_altura(s);
}
```

Algoritmo 2.17 *Rotación simple derecha_ derecha sobre un nodo apuntado por r con hijo derecho no vacío.*

Estas operaciones son la esencia de los árboles binarios de búsqueda AVL. Sin embargo, sólo se han resuelto los desequilibrios más sencillos. Antes de poderlas utilizar hay que resolver aquellos desequilibrios producidos por un hijo izquierdo de un hijo derecho, o a la inversa, un hijo derecho de un hijo izquierdo.

En estos dos casos se procederá haciendo primero una rotación simple sobre el primero de los hijos en la rama que produce el desequilibrio. En la Figura 2.22 se puede observar el estado inicial de los subárboles desequilibrados, y el par de rotaciones, o rotaciones dobles, a las que hay que someter esos árboles para equilibrarlos.

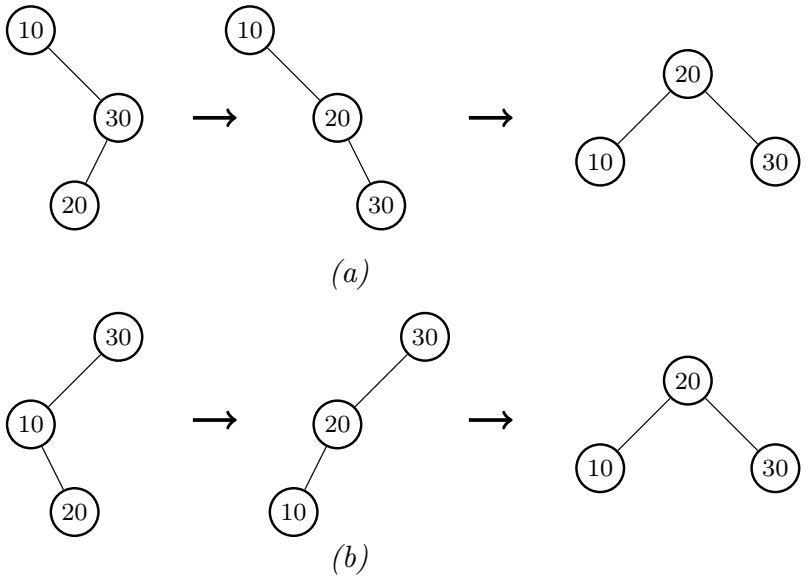


Figura 2.22: *(a) Rotación doble derecha_ izquierda; (b) Rotación doble izquierda_ derecha.*

Es fácil pues, suponer que las implementaciones de estas funciones tienen el aspecto que se puede comprobar en los Algoritmos 2.18 y 2.19 respectivamente.

```
void derecha_izquierda(nodo*& r) {
    izquierda_izquierda(r→hd);
    derecha_derecha(r);
}
```

Algoritmo 2.18 *Rotación doble derecha_ izquierda sobre un nodo apuntado por r con hijo derecho no vacío que tiene hijo izquierdo no vacío.*

```
void izquierda_derecha(nodo*& r) {
    derecha_derecha(r→hi);
    izquierda_izquierda(r);
}
```

Algoritmo 2.19 *Rotación doble izquierda_ derecha sobre un nodo apuntado por r con hijo izquierdo no vacío que tiene hijo derecho no vacío.*

Igualmente, las rotaciones simples tienen la eficiencia de las rotaciones dobles.

Está bien claro que los tiempos requeridos serán, aproximadamente, uno el doble del otro, cosa que queda absorvida por la constante oculta de la notación.

Inserción en los árboles AVL

Finalmente, una vez provistos de las cuatro funciones de rotación descritas, vamos a echar un vistazo a la nueva función de inserción para este tipo de árboles binarios de búsqueda.

El Algoritmo 2.20 no es más que la inserción implementada para los árboles binarios de búsqueda en el Algoritmo 2.9, más, adicionalmente, un control del equilibrio en cada uno de los nodos que va de la hoja acabada de insertar, hasta la raíz, por su rama. En las dos sentencias alternativas se comprueba que la diferencia de alturas entre los dos hijos de cada uno de los nodos de la rama sea correcta.

```
void _insertar(nodo*& r, elemento e) {
    if (r == NULL) {
        r = new nodo;
        r→e = e;
        r→hi = NULL;
        r→hd = NULL;
        ++n;
    }
    if (e.clave < r→e.clave) {
        if (calcular_altura(r→hi) - calcular_altura(r→hd) == 2) {
            if (e.clave < r→hi→e.clave) izquierda_izquierda(r);
            else izquierda_derecha(r);
        }
        calcular_altura(r);
    }
    else if (e.clave > r→e.clave) {
        if (calcular_altura(r→hd) - calcular_altura(r→hi) == 2) {
            if (e.clave < r→hd→e.clave) derecha_derecha(r);
            else derecha_izquierda(r);
        }
        calcular_altura(r);
    }
    else r→e = e;
}
```

Algoritmo 2.20 *Inserción en un árbol binario de búsqueda AVL.*

Observad que el control está justo después de la llamada recursiva. Eso

significa decir que cuando finalmente la recursividad llegue a las hojas, casos triviales, se retornará. Y entonces, después de retornar de la llamada recursiva, se irá controlando el equilibrio para el padre del nuevo nodo, para el padre del padre, y así hasta la raíz.

De la misma manera que en el Algoritmo 2.9 se ha tomado una decisión de diseño a partir de la que se rechazaban las inserciones de claves ya existentes, en este caso se ilustra la decisión contraria. En el Algoritmo 2.20, cuando se pretende insertar un elemento con clave ya existente en el diccionario, se copia su descripción, machacando la anterior. En definitiva, se trata de comprender que el comportamiento de estas funciones frente situaciones de extremo, hay que decidirlas en la implementación por alguna opción bien determinada.

La eficiencia de la operación de insertar en un AVL es $\Theta(log(n))$, ya que ahora sí que podemos asegurar que el árbol será equilibrado.

En el código que acompaña este libro, las implementaciones de estas operaciones son difíciles de comprobar. Para aquellos que deseen hacerlo, deberán de utilizar un depurador. Sólo así se podrá ver que la representación de las estructuras es la correcta.

Operaciones privadas para la eliminación en los árboles AVL

Así como se ha añadido código para la operación de inserción con el fin de mantener el equilibrio después de cada nuevo elemento, ahora hacen falta algunas modificaciones del cuerpo de la operación de eliminación. Primero, un par de operaciones auxiliares. Estas operaciones pueden ser llamadas en cualquier momento que se desee equilibrar el árbol.

```
void equilibra_izquierda(nodo*& r)
{
    if (calcular_altura(r→hi)-calcular_altura(r→hd)==2) {
        int hid = calcular_altura(r→hi→hd);
        int hii = calcular_altura(r→hi→hi);
        if (hid - hii == 1) izquierda_derecha(r);
        else izquierda_izquierda(r);
    }
    else calcular_altura(r);
}
```

Algoritmo 2.21 *Operaciones de equilibrio en un árbol AVL desequilibrado por la izquierda.*

En el Algoritmo 2.21 se muestra la implementación de *equilibra_ izquierda()*, una operación útil para cuando un nodo tiene nietos por la izquierda sin tener hijos por la derecha. Entonces, sabiendo que el hijo izquierdo tiene más altura que el derecho en dos unidades, decide cuál de los hijos de su hijo izquierdo provoca el desequilibrio. Y según el caso, se llama a una rotación simple, o doble.

En el Algoritmo 2.22 se puede ver el caso simétrico.

```
void equilibra_derecha(nodo*& r)
{
    if (calcular_altura(r→hi)-calcular_alcada(r→hd)==2) {
        int hdi = calcular_altura(r→hd→hi);
        int hdd = calcular_altura(r→hd→hd);
        if (hdi - hdd == 1) derecha_izquierda(r);
        else derecha_derecha(r);
    }
    else calcular_altura(r);
}
```

Algoritmo 2.22 *Operaciones de equilibrio en un árbol AVL desequilibrado por la derecha.*

De hecho, se hubieran podido hacer las inserciones del Algoritmo 2.20 exactamente igual a las del Algoritmo 2.9, y después llamar a las funciones equilibra_izquierda() y equilibra_derecha() de los Algoritmos 2.21 y 2.22. Lo que pasa, es que cuando se hace una inserción ya se sabe qué desequilibrio puede ser necesario enfrentar, y por tanto ya hay medio trabajo hecho.

Eliminación en los árboles AVL

En la Sección 2.2.3 se ha visto en detalle el razonamiento que se sigue en el procedimiento de eliminación de nodos en los árboles binarios de búsqueda. Aparte de las operaciones internas de equilibrio también hace falta adecuar la función de *sacar_ maximo* del Algoritmo 2.11 de la página 88. Está claro que al sacar un nodo, en este caso el de valor de clave máximo, se puede producir un desequilibrio. Así pues, en el Algoritmo 2.23 hay una implementación de la función de sacar el máximo para el caso de los árboles AVL. Esta operación se efectúa en un tiempo $\Theta(h)$.

```
nodo* sacar_maximo(nodo*& r)
{
    if (r→hd) {
        nodo* s = sacar_maximo(r→hd);
        equilibra_derecha(r);
        return s;
    }
    else {
        nodo* s = r;
        r = r→hi;
        return s;
    }
}
```

Algoritmo 2.23 *El nodo con clave máxima de un árbol AVL no vacío se quita del árbol y se retorna el apuntador que lo señala.*

En el Algoritmo 2.24 se muestra el código para la eliminación en árboles AVL.

```
bool _eliminar(nodo*& r, elemento e) {
    if (r == NULL) return false;
    if (e.clave < r→e.clave) {
        _eliminar(r→hi,e);
        equilibra_izquierda(r);
    }
    else if (e.clave > r→e.clave) {
        _eliminar(r→hd,e);
        equilibra_derecha(r);
    }
    else {
        nodo* s = r;
        if (r→h==0) r = NULL;
        else if (r→hi == NULL) r = r→hd;
        else if (r→hd == NULL) r = r→hi;
        else {
            nodo* m = sacar_maximo(r→hi);
            m→hi = r→hi; m→hd = r→hd; r = m;
            equilibra_derecha(r);
        }
        delete s; --n;
    }
    return true;
}
```

Algoritmo 2.24 *Eliminación en un árbol binario de búsqueda AVL.*

La función del Algoritmo 2.24 realiza la misma tarea que la del procedimiento mostrado en el Algoritmo 2.12 con el reequilibrio adicional de cada nodo de la rama que va desde el nodo eliminado hasta la raíz. La eficiencia de este procedimiento es $\Theta(log(n))$.

Finalmente, después de una sucesión de implementaciones para los diccionarios, hemos conseguido una estructura de datos que, aunque es un pelo sofisticada, sí que satisface las espectativas que se habían puesto para una estructura especializada en buscar elementos en una colección. Difícilmente puede encontrarse una implementación más rápida para realizar esta operación.

2.3 Colas de Prioridad

Una estructura de datos muy interesante para multitud de aplicaciones sería aquella que, con algún tipo de colección de objetos, pudiera proporcionar, tan pronto como fuera posible, el objeto que más nos interese. Para eso, deberíamos cuantificar el interés que tenemos en cada uno de los objetos.

A este interés cuantificado, es decir, numérico, lo denominamos *prioridad*.

Partimos de la idea de una estructura de datos que nos permita obtener valores que hayamos introducido de uno en uno, como en las listas, o en las pilas. Entonces, el comportamiento que se le pide a esta estructura debe permitir que cuando se inserte un objeto con prioridad alta, sólo por ese motivo, sea capaz de adelantarse a los de prioridad inferior. Y por tanto, pueda ser proporcionado al exterior antes que todos esos otros objetos.

Para llevar a término esa estructura, formalizamos la siguiente definición.

Definición 2.7 Cola de Prioridad. *Estructura de datos contenedora de ítems con prioridades especializada en obtener el más prioritario.*

La prioridad es un concepto que no está íntimamente ligado al crecimiento numérico cuando se interpreta. Hay sistemas para los cuales la máxima prioridad se codifica con el valor 1, de manera que cuando más alto sea el valor menos prioritario es el elemento. En el caso de los campeonatos de futbol, siempre se considera que los equipos de primera división son mejores que los de segunda, aunque 2 es mayor que 1. Más ejemplos, para todos aquellos casos en los que se utiliza el término *categoría*, ocurre exactamente igual. Cuando se dice que un coche es de primera categoría, o que un vino es de primera categoría, siempre significa que es lo mejor.

Probablemente se ha escogido la palabra "prioridad" para no comprometer el sentido de cuál es más alta o más baja. De manera que la estructura de datos que nos disponemos a analizar tiene dos posibles implementaciones: Aquellas colas en las que la prioridad más alta se corresponde con el valor más alto de prioridad, y a la inversa, cuando la prioridad más alta viene codificada con el valor menor.

En esta sección consideraremos siempre la primera interpretación. La prioridad más alta es la que tiene el valor más alto de prioridad. No obstante, todo lo que se diga en adelante puede ser fácilmente adaptado al caso contrario (tan sólo cambiando los '$<$' por '$>$'). En otras palabras, nos disponemos a implementar estructuras de datos contenedoras de ítems con prioridades, especializadas en la operación de extraer el máximo, pero fácilmente podríamos hacer la otra versión, extraer el mínimo.

Observad que estemos diciendo "extraer". En la Definición 2.7 decía "obtener". En general, para las colas de prioridad se asume que la consulta del máximo provoca la extracción de este elemento de la colección. Eso es así, como en otras estructuras, porque de esta manera han sido más útiles. Es una decisión de diseño. De esta forma, además, podremos utilizar estas colas para ordenar una colección de n números de una manera trivial. Se tratará de hacer n extracciones seguidas, del máximo. Así obtendremos los elementos ordenados.

Con colas de prioridad se hace la gestión de errores en un sistema operativo, o de incidencias en cualquier tipo de sistema. Sería cosa buena que en las urgencias de los hospitales asignasen una prioridad a cada nuevo enfermo que llegara, de manera que el orden en ser atendidos dependiera exclusivamente de ese indicador. Para modelar esa cola sería necesaria una estructura como la que nos disponemos a implementar.

Y un último ejemplo, la ordenación cronológica. Cuando tenemos una colección de acontecimientos, o eventos que proceden de distintas fuentes y tardan tiempos impredictibles en recorrer el canal de comunicación, normalmente se recibirán desordenados en el sistema central. Para obtener una lista de los eventos ocurridos por orden cronológico, se puede considerar el tiempo como una prioridad, de manera que el más prioritario sea el más antiguo. Implementándolo de ese modo, obtendríamos todos los acontecimientos notificados por orden.

2.3.1 Implementaciones sencillas

En la sección anterior, cuando hablábamos de los diccionarios, hemos entrado en un nivel de detalle para las estructuras que soportan las implementaciones sencillas, que ahora ya no merece la pena. Lo hemos hecho así para repasar un poco las estructuras a las que nos referíamos cuando decíamos implementaciones sencillas, y cómo funcionaban. Por lo que respecta a las colas de prioridad, como se puede observar en la Tabla 2.1 nos limitamos a mencionar las eficiencias según las distintas implementaciones, y no vamos más allá. Se asume que el lector no tendría ninguna dificultad en implementar una cola de prioridad en un vector o una lista.

Como operaciones características se considera obtener el elemento de máxima prioridad, *getmax()*, e insertar un elemento, *insert()*.

La decisión de mantener el orden en las estructuras como el vector o la lista es importante para las colas de prioridad. Así pues, en la Tabla 2.1, para acotar los tiempos de respuesta de estas dos operaciones, se distingue entre los dos casos.

Tan sólo para las implementaciones en vectores, analizamos brevemente las cotas expresadas en la Tabla 2.1. Supondremos que el vector ordenado lo está

	getmax()	insert()
vector desordenado	$\Theta(n)$	$\Theta(1)$
vector ordenado	$\Theta(1)$	$\Theta(n)$
lista desordenada	$\Theta(n)$	$\Theta(1)$
lista ordenada	$\Theta(1)$	$\Theta(n)$

Tabla 2.1: *Comparación de las eficiencias de insertar y de obtener el máximo en las implementaciones sencillas de las colas de prioridad.*

crecientemente, o sea que el máximo se encuentra en la posición n. En este análisis, además de la notación asimptótica, también se identifican los casos. Observad que para hablar de los casos hacemos continuas referencias a la palabra "valor".

- Que un vector desordenado tarde $\Theta(n)$ a extraer el máximo es debido a que es necesario un recorrido lineal de todo él para identificar este máximo, $\Theta(n)$ en cualquier caso, o sea, para cualquier posición que ocupe el valor máximo. Además, después, para borrarlo es necesario desplazar todos los elementos siguientes, que también tarda $\Theta(n)$ en el caso peor, o sea, cuando el valor máximo estaba en la primera posición del vector.

- En un vector desordenado insertaremos siempre en la última posición, $\Theta(1)$ en cualquier caso, o sea, independientemente del valor del nuevo elemento y de los que haya en el vector.

- Para el caso del vector ordenado, obtener y extraer el máximo es acceder a la última posición, y retornar el elemento liberando el espacio que ocupa. Eso significa $\Theta(1)$ en cualquier caso, o sea, independientemente del valor del máximo y de los otros valores del vector.

- Insertar en un vector ordenado puede requerir desplazar todos los elementos en el peor de los casos, $\Theta(n)$, o sea, cuando el valor del nuevo elemento es menor que todos los otros del vector, es decir, el primero.

Razonamientos similares justificarían las eficiencias para las listas.

Definitivamente, estas eficiencias no nos satisfacen. Pasemos pues a diseñar estructuras más eficientes para la implementación de las colas de prioridad.

2.3.2 Heaps

Un heap es una estructura de datos que sirve para implementar las colas de prioridad. Físicamente se soporta sobre un vector y realiza las operaciones en el mismo espacio, de manera que la eficiencia espacial de la estructura, $\Theta(n)$, se puede considerar óptima.

Representamos los heaps en árboles *semicompletos*. Un árbol semicompleto es aquél que todos los niveles están llenos excepto, quizás, el último. De manera que todas las hojas están en los dos últimos niveles. En el penúltimo por la parte derecha, y en el último, por la parte izquierda, tal como se ve en la Figura 2.23. Se trata, pues, de una estructura más rígida que aquélla que teníamos para los árboles de búsqueda. Ahora, podemos guardar la estructura en un vector estático. Sin embargo, no podemos mover contenidos de elementos, y por tanto se requiere acompañar el heap con alguna estructura que almacene efectivamente la información, y gestionar las operaciones a partir del heap que apunta a los elementos de la cola, es decir, a todo aquello que no son las prioridades.

Estamos hablando de árboles binarios. Luego, en el nivel k habrá 2^k nodos cuando esté lleno. De todo ello, a diferencia de los árboles binarios de búsqueda, resulta una representación única para cada cantidad de elementos n que haya en el heap.

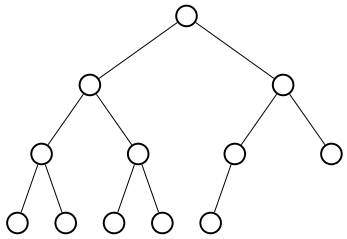


Figura 2.23: *Representación de un heap en un árbol semicompleto.*

Para los valores de las prioridades, independientemente de la topología de la representación del heap, tenemos lo que llamamos *condición de heap*. Esta condición se define tanto sobre el árbol que recién hemos visto, como sobre el vector en el que finalmente acabaremos implementando este árbol. Por lo que respecta al árbol semicompleto, cumplir la condición de heap significa que las prioridades de los nodos padre son más altas que las de los hijos. No es difícil darse cuenta pues que el valor máximo de prioridad se encontrará en la raíz. La definición formal la establecemos sobre el vector.

Definición 2.8 Condición de heap. *Se dice que un vector T de n elementos ordenables satisface la condición de heap si para cualquier índice $i \in [1, n/2]$, ocurre que*

$$T[i] > T[2i] \text{ y que } T[i] > T[2i+1].$$

Rigurosamente, la Definición 2.8 da la condición necesaria y suficiente que debe satisfacerse en un *max-heap*. Es decir, el que estamos considerando. En el caso contrario, cuando se decide que la máxima prioridad es la que tiene el valor más bajo se dice que tenemos un *min-heap*. En el caso de un min-heap se debe cumplir que $T[i] < T[2i]$ y que $T[i] < T[2i+1]$ para satisfacer la condición de heap. Y claro, la operación característica de la estructura no es obtener el máximo, sinó obtener el mínimo, *getmin()*.

Por otro lado, los signos '$>$' de la definición pueden ser sustituidos por '$\geq$' cuando permitimos valores de prioridad repetidos en la estructura.

Bien, cabe destacar que la condición de heap no es categórica, es una teoría abierta. Heaps equivalentes se pueden representar de distintas maneras. Tenemos un margen de liberdad en el orden de los hijos. No hay ninguna restricción entre los dos hijos de un mismo padre ni entre todos los nodos de un mismo nivel.

Vamos a por la transformación del árbol en vector. Observad que de la condición de heap, se desprende que la posición $i = 0$ debe quedar vacía. En los lenguajes que los vectores comienzan en el índice $i = 0$, será lo correcto no hacer uso de este elemento, declarando siempre los heaps como vectores $T[1 + n]$. De esta manera se ha hecho en el código que se proporciona con este libro.

Pondremos, pues, la raíz en la posición del índice $i = 1$. A partir de aquí, para cualquier índice $i \in [1, n/2]$, tendremos el nodo correspondiente a su hijo izquierdo en la posición indexada por $2i$, y su hijo derecho al lado, en la $2i + 1$.

Se puede ver una instancia correspondiente a un heap concreto en la Figura 2.24. En la parte izquierda, Figura 2.24*(a)*, se muestra la representación en forma de árbol. En la parte derecha, Figura 2.24*(b)*, se puede observar el vector donde lo almacenamos, satisfaciendo la condición de heap. El contenido del vector no es más que el recorrido por niveles del árbol.

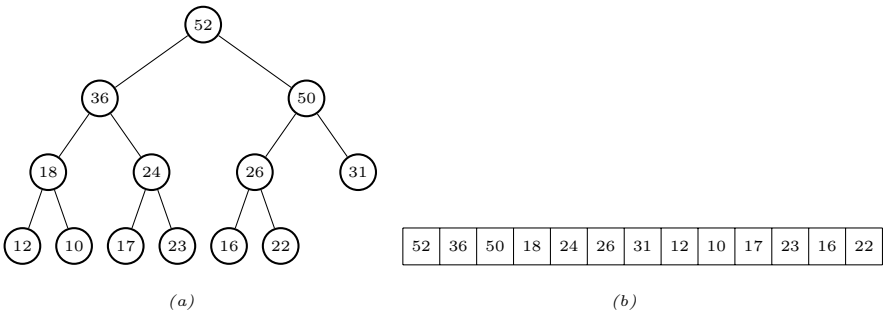


Figura 2.24: *Representaciones de un heap:(a) Arborescente; (b) Vectorial.*

Como se puede deducir de la definición de heap, un vector ordenado decrecientemente satisface la condición.

Notad, finalmente, que la segunda mitad del vector está totalmente desordenada, y a medida que vayamos aproximándonos a la raíz, a la posición 1, cada vez está más ordenado. Con todo, se provoca que a la hora de representarnos mentalmente un heap tengamos que imaginar una estructura híbrida como la que se muestra en la Figura 2.25.

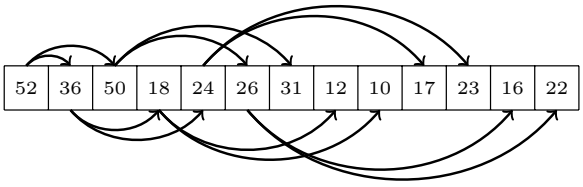


Figura 2.25: *Representación arborescente en una disposición vectorial.*

Como veremos seguidamente, esta estructura dará eficiencias logarítmicas para las operaciones de las colas de prioridad.

2.3.3 Implementación de los Heaps

Para el caso de los diccionarios, cuando los implementábamos en árboles binarios de búsqueda, había mucho movimiento de apuntadores, y poco de contenidos. Eso nos permitía tener físicamente las descripciones de los elementos en los nodos. Ahora no será así.

Como se verá en la implementación de las operaciones, el movimiento de ítems por la estructura es intensa. Y no conviene arrastrar grandes cantidades de memoria con frecuencia. Por esta razón, así como antes partíamos de una estructura que llamábamos *elemento*, para los heaps partimos de una estructura más ágil, que no lleva la descripción de los elementos en su interior.

Lo que haremos en este caso será suponer que los elementos con sus descripciones se encuentren en algún lugar, ordenados por orden de clave. Aquí, definimos una estructura que llamamos *ítem*, mostrada en el Algoritmo 2.25, y que tan sólo guarda un valor numérico, en el campo *datos*, correspondiente al índice del elemento completo en el lugar donde se guarden, y la prioridad con la cual se trabajará en el heap. Quizás hubiera sido más realista poner un apuntador a los datos, más que un entero para indexarlos. Se ha hecho así con la intención de facilitar la comprensión de la materia, ya que al fin y al cabo, el campo *datos* de la estructura del Algoritmo 2.25 no se utilizará funcionalmente en ninguno de los procedimientos.

```
struct item {
    int datos;
    int prioridad;
};
```

Algoritmo 2.25 *Estructura básica para los elementos del heap.*

```
#include "item.h"

class cola_de_prioridad_heap // implementa un max_heap
{
private:
    item *cola;
    int n;

    void promocionar(int k) {
        while (k>1 && cola[k].prioridad > cola[k/2].prioridad) {
            item aux = cola[k]; cola[k] = cola[k/2]; cola[k/2] = aux;
            k = k/2;
        }
    }

    void hundir(int k, int n) {
        while (2*k <= n) {
            int j = 2*k;
            if (j<n && cola[j].prioridad < cola[j+1].prioridad) j++;
            if (cola[k].prioridad > cola[j].prioridad) break;
            item aux = cola[k]; cola[k] = cola[j]; cola[j] = aux;
            k = j;
        }
    }

public:
    cola_de_prioridad_heap(int N) {cola = new item[1+N]; n = 0;}
    ~cola_de_prioridad_heap() {delete [ ] cola;}

    bool insertar(item i) {
        if (n == 1 + sizeof(cola)) return false;
        cola[++n] = i;
        promocionar(n);
        return true;
    }

    bool getmax(item& i) {
        if (n==1) return false;
        item aux = cola[1]; cola[1] = cola[n]; cola[n] = aux;
        hundir(1,n-1);
        i = cola[n--];
        return true;
    }
};
```

Algoritmo 2.26 *Implementación de las colas de prioridad en un heap.*

En el Algoritmo 2.26 se describe la clase *cola_de_prioridad_heap*. Representamos los heaps en vectores de ítems como el de la estructura mostrada en el Algoritmo 2.25. Se trata de una estructura semiestática, es decir, dejaremos que sea el usuario (el código que utiliza la estructura) quien nos delimite el espacio que hay disponible en el momento de la creación de la cola.

Como se puede ver en el Algoritmo 2.26, como variable miembro de la clase definimos un apuntador a un ítem, *cola*, que luego tratamos como un vector. Su dimensión se establece en el constructor. Ésta es una manera muy extendida de gestionar la memoria dinámica. Haciendo una sola petición. La máquina virtual de java actúa de ese modo. Tiene el inconveniente de que no ajusta muy finamente la cantidad de memoria disponible a la necesaria. Pero, por otro lado, ahorra tiempo. Las solicitudes frecuentes de memoria ralentizan los procesos.

Estudiemos ahora, en detalle, cada una de las operaciones del Algoritmo 2.26, tanto públicas como privadas.

Construcción y Destrucción

Se trata de que sea cual sea el método con el que construimos una cola de prioridad implementada en un heap, el resultado después del constructor debe satisfer la condición de heap.

Para la creación hay dos estrategias bien distintas:

- *top-down*: La construcción de un heap más sencilla de comprender es con el método descendiente. Aunque por otra parte, la más ineficiente. Dado el vector inicialmente vacío, se trata simplemente de realizar las n inserciones. Finalmente, por la manera como se hacen esas inserciones, el resultado obtenido es un vector que satisface la condición de heap.
- *bottom-up*: Para poder crear un heap ascendientemente hay que conocer a priori los elementos que contendrá. Es decir, sólo podemos crear un heap con esta técnica cuando podamos partir de un vector con los elementos que finalmente deben quedar contenidos en el heap.

Lo que se hace en el Algoritmo 2.26 es una creación top-down, ya que no se dispone de los elementos que llenarán la estructura. En la Sección 2.3.4 con el método de ordenación con heaps, heapsort, se profundiza en la creación bottom-up de heaps.

Como se puede ver también, por el único constructor que hay disponible, es que para crear una cola de prioridad hay que dar una cota superior al espacio que se prevé necesitar. En el constructor se reserva la memoria, y no se liberará hasta al destructor.

Operaciones privadas para la extracción del máximo y la inserción

Hay una clara analogía entre la estructura arborescente de los heaps presentada en la Figura 2.24 y cualquier estructura jerárquica. Eso nos es útil a la hora de denominar a las operaciones internas. Estas dos operaciones servirán para subir y bajar niveles dentro de la estructura.

Tal como se puede observar en el Algoritmo 2.26, comenzamos proveyéndonos de unas operaciones internas, promocionar y hundir, que después utilizamos al implementar la extracción del máximo y la inserción.

Promocionar

Utilizaremos la palabra *promocionar* para referirnos al procedimiento que implementa la ascensión de un ítem por la estructura arborescente.

En el Algoritmo 2.27, el parámetro entero k es el índice del elemento de la cola que nos disponemos a reubicar más arriba en el árbol. Como precondición, esta operación supone que la k es inferior o igual al número de elementos de la cola, n. De todas formas, la n no se utiliza en esta operación.

```
void promocionar(int k) {
    while (k>1 && cola[k].prioridad > cola[k].prioridad/2) {
        item aux = cola[k]; cola[k] = cola[k/2]; cola[k/2] = aux;
        k = k/2;
    }
}
```

Algoritmo 2.27 *Operación interna para ascender en la estructura de prioridades.*

En detalle,

1a. línea: Cabecera del procedimiento. Tan sólo se recibe, por valor, el índice al vector *cola* del elemento que probablemente deba ascender por la jerarquía.

2a. línea: Esquema clásico de búsqueda secuencial, *mientras no final y no encontrado*. La k toma valores de la sucesión $k_0/2^i$, siendo k_0 el valor inicial de k, e i el número de iteración. Esta sucesión converge al valor 1. La condición de final, pues, es $k = 1$, y por tanto, la $k > 1$ significa *no final*. Además, estamos buscando el lugar donde ha de ir k. Eso es, el lugar donde la prioridad del padre sea mayor que la suya. Por tanto, mientras

la prioridad del elemento k sea mayor que la de su padre (el elemento $k/2$), es que todavía el lugar está *no encontrado*.

3a. línea: Seguros de que el elemento k tiene la prioridad mayor que su padre, vamos restaurando la condición de heap haciendo intercambios. En esta línea se hace un intercambio en cada iteración.

4a. línea: Una vez restaurado el nivel más alto (más cercano a las hojas del árbol), pasamos a un nivel inferior de cara a restaurar la condición de heap hasta la raíz.

Observad que para ascender sólo hay un camino posible, la rama de la cual cuelga el nodo que se promociona. De ahí que no haya ningún *if* en el Algoritmo 2.27. La operación de promocionar requiere un tiempo perteneciente a $\Theta(log(n))$, es decir, el número de iteraciones que se realizarán en el caso peor, o sea, cuando un elemento con una prioridad muy alta, la máxima, se encuentre (por la razón que sea), en la mitad final del vector, la de los índices más altos con valores más bajos. Para comprobarlo, hay considerar la sucesión de valores que toma, $k_i = k_0/2^i$, vista en el análisis de la segunda línea.

Hundir

La operación inversa a promocionar es la de *hundir*. Utilizamos este término para denominar el procedimiento que servirá para hacer descender un ítem a través de una rama del árbol. En este caso, a diferencia de la promoción, se encuentra en una disyuntiva en cada paso.

En el Algoritmo 2.28, se recibe el índice k del elemento que nos disponemos a hundir, y n como el límite del espacio donde podemos reubicarlo. Como precondición tenemos que $k \leq n$, siendo n el número de ítems en la cola de prioridad.

```
void hundir(int k, int n) {
    while (2*k <= n) {
        int j = 2*k;
        if (j<n && cola[j].prioridad < cola[j+1].prioridad) j++;
        if (cola[k].prioridad > cola[j].prioridad) break;
        item aux = cola[k]; cola[k] = cola[j]; cola[j] = aux;
        k = j;
    }
}
```

Algoritmo 2.28 *Operación interna para descender en la estructura de prioridades.*

En detalle,

1a. línea: Cabecera del procedimiento. Se recibe por valor el índice k del elemento a hundir, y n, el número de elementos que hay en el heap.

2a. línea: El bucle tan sólo controla la condición de final. No obstante, esto no es un recorrido completo. No se está siguiendo un esquema, *mientras no final*, ya que dentro del bucle hay un break para el caso de *encontrado*. Es decir, como antes, seguimos en un esquema de *mientras no final y no encontrado*. Lo que pasa, es que la condición de *encontrado* es más compleja que antes. Un padre tiene dos hijos, y la condición de *encontrado* requiere dos comparaciones. Para no hacer la condición del bucle demasiado pesada, se ha preferido dejar la condición de *final* en la cabecera del bucle, y poner un *break* en el interior para el caso de *encontrado*. La k tomará valores de la sucesión $k_0 * 2^i$, siendo k_0 el valor inicial de k, e i el número de iteración. Esta sucesión crece indefinidamente. Por tanto, podemos asegurar que en algún momente $2k > n$ y se saldrá del bucle.

3a. línea: Una vez dentro del bucle, estamos seguros que k tiene, al menos, hijo izquierdo. Declaramos un nuevo índice, j, y hacemos que apunte al hijo izquierdo del nodo que estamos hundiendo.

4a. línea: Hemos entrado en el bucle porque $2k \leq n$. Eso nos asegura que k tiene hijo izquierdo. Cuando $2k = n$, entonces significa que k tiene hijo izquierdo pero no hijo derecho. Ahora, j apunta al hijo izquierdo de k. La primera condición de esta sentencia alternativa $j < n$ nos asegura que k, a parte de hijo izquierdo tenga hijo derecho. O sea, sabiendo que k tiene hijo izquierdo, esta línea provoca que, si también tiene hijo derecho, entonces j apunte a la máxima prioridad entre los dos hijos.

5a. línea Comparamos la prioridad de k con la máxima de entre sus dos hijos apuntado por j. Si la prioridad de k es mayor que la máxima de sus hijos, entonces ya hemos encontrado su posición. No es necesario que sigamos hundiendo, y salimos del bucle.

6a. línea: Hacemos el intercambio de ítems entre el indicado por k y el que tenga máxima prioridad de sus hijos, indicado por j.

7a. línea: Hacemos descender la k para continuar hundiendo hasta que llegue a la segunda mitad del vector, $2k > n$, en el peor de los casos, o tenga más prioridad que sus dos hijos y salga por el *break*.

En el tipo de asignaciones que se hace a j en la cuarta línea hay alguna cosa poco elegante. Es como cuando para calcular el máximo entre dos números, a y b, procedemos diciendo $max = a$, y *si* $(a < b)$ *entonces* $max = b$. La falta de elegancia es la serialización de una asignación paralela por naturaleza, o, dicho de otra manera, implementar en un código no commutativo una operación tan commutativa como es el máximo.

Bien, en cualquier caso es correcta, y en el Algoritmo 2.28 tenemos un buen uso de ello. Es decir, cuando pasemos del padre al hijo nos quedamos en primera instancia con el hijo izquierdo y si después resulta de más alta prioridad el hijo derecho, entonces cogemos esta nueva rama. También se puede dar el caso que por una de las dos ramas no tengamos que bajar, y por la otra sí. Es decir, que la prioridad de k sea más alta que la de su hijo izquierdo y más baja que la de su hijo derecho. En este caso, se continúa hundiendo por la rama derecha.

La operación de hundir requiere también un tiempo perteneciente a $\Theta(log(n))$, es decir, el número de iteraciones que se realizarán en el caso peor. Como antes, para comprobarlo, hay que considerar la sucesión de valores que toma k_i, vista en el análisis de la segunda línea, $k_i = k_0 * 2^i$.

Operaciones características de las colas de prioridad en un heap

Las dos operaciones que caracterizan las colas de prioridad, tanto insertar como obtener el máximo, funcionan en dos pasos. En el primero rompen la condición de heap, cosa que es fácil, y en el segundo la restauran utilizando alguna operación interna de las descritas en las dos secciones anteriores, cosa no tan fácil.

Inserción

Como se puede observar en el Algoritmo 2.26, una vez implementada la operación de promocionar, insertar en un heap es tan sencillo como añadir el nuevo ítem en la última posición, y ascenderlo por la estructura hasta donde le toque. En el Algoritmo 2.29 se repite el código que implementa la inserción.

```
bool insertar(item i) {
    if (n == sizeof(cola) - 1) return false;
    cola[++n] = i;
    promocionar(n);
    return true;
}
```

Algoritmo 2.29 *Inserción de un ítem en una cola de prioridades implementada en un heap.*

La función de inserción, como en todos los casos anteriores, retorna un booleano para avisar al módulo usuario de si efectivamente se ha insertado

el ítem. En caso de que el módulo que pide esta inserción no haya declarado suficiente espacio en el momento de la creación de la cola de prioridad, aquí podríamos optar por hacer una llamada al sistema pidiendo más memoria, pero por sencillez, se ha decidido retornar el valor *falso*.

Una vez está claro que disponemos de la memoria necesaria, incrementamos el número de elementos n, y ponemos en la última posición del vector el nuevo elemento. En este momento puede haberse violado la condición de heap. Seguidamente hacemos la llamada para que el ítem ascienda si su prioridad lo requiere, haciendo uso de la operación privada promocionar. Al final de la inserción, la condición de heap quedará restaurada.

La eficiencia de la inserción viene dominada por el procedimiento de promocionar, y por tanto se queda en $\Theta(log(n))$ en el caso peor, es decir, cuando el nuevo elemento tenga la prioridad más alta de todo el heap y por tanto deba ser promocionado hasta la raíz.

Obtención del máximo

Como se puede observar en el Algoritmo 2.26, una vez implementada la operación de hundir, obtener el ítem de máxima prioridad en un heap es tan fácil como retornar la raíz. Para quitarla, sin embargo, la intercambiamos de posición con el ítem de la última hoja. Así pues, lo hacemos todo in situ. Como consecuencia se ha roto la condición de heap. Se debe hundir esta hoja recién colocada en la raíz hasta donde le toque (que como mucho será $n-1$), para restaurarla. En el Algoritmo 2.30 repetimos la implementación del código de la operación *getmax()*.

```
bool getmax(item& i) {
    if (n==0) return false;
    item aux = cola[1]; cola[1] = cola[n]; cola[n] = aux;
    hundir(1,n-1);
    i = cola[n--];
    return true;
}
```

Algoritmo 2.30 *Obtención del ítem de máxima prioridad en una cola de prioridades implementada en un heap.*

En la primera línea, estamos comprobando que la cola no esté vacía. Si así fuera, retornaríamos *falso*. En otro caso, siempre existirá un valor máximo. En la segunda línea, intercambiamos los ítems primero, que es el máximo que nos piden, y el último, que será, en principio, uno de los elementos con una prioridad

baja. En este momento se ha roto la condición de heap, ya que en la primera posición del vector, que representa la raíz, hay un elemento con menos prioridad que alguno de sus hijos, y eso es, precisamente, romper la condición de heap. Entonces, se hunde limitando hasta donde puede caer a $n-1$, que será el nuevo número de elementos de la cola.

Fijaos bien, que aunque la cola se piense que sólo tiene el nuevo número de elementos, $n-1$, realmente, escondido tras éstos nos quedará el máximo, y seguirá en esta posición a no ser que se haga una nueva inserción. De esta manera, aprovechamos el mismo espacio, y si se hicieran n extracciones de máximo seguidas, el vector quedaría ordenado crecientemente.

Para la operación de extraer el máximo, como antes, también tenemos la eficiencia dominada por el procedimiento de hundir. Por tanto, también es $\Theta(log(n))$. Lo que pasa es que ahora no sólo en el caso peor, sinó en todos los casos, ya que siempre pasará que la hoja acabada de intercambiar con la raíz deba ser hundida toda la altura del árbol.

2.3.4 Heapsort

Después de haber visto el funcionamiento de una cola de prioridad parece claro de qué manera las podemos utilizar para ordenar un vector, haciendo n llamadas a extraer el máximo. Si sólo fuera eso, no valdría la pena dedicar una sección al algoritmo de ordenación *heapsort*. No. Hay alguna cosa más.

Ahora, a diferencia de la sección anterior, suponemos que queremos ordenar elementos que ya tenemos en un vector, y aquí radica la diferencia. Tenemos todos los elementos que queremos insertar en la cola de prioridad antes de empezar, de manera que ya sabemos, antes de crear el heap, qué elementos habrá. Y por tanto cuántos.

Cuando se trata de crear un heap para un conjunto de elementos que a priori se tienen en un vector, entonces hay una manera más eficiente que partiendo de cero y haciendo las n inserciones. A esta manera la llamamos bottom-up, o ascendente. Y a la rutina que la implementa, *crear_heap*.

En el Algoritmo 2.31, *crear_heap* trabaja sobre el vector de ítems que la rutina *heapsort* ha construido a partir del vector de entrada que se desea ordenar. Entonces, in situ, se dedica a hundir los elementos de la primera mitad del vector por orden decreciente.

Y ahora atención,

$$T_{crear\_heap}(n) = \Theta(n).$$

Sin duda sorprendente. Bien, que fuera $\Omega(n)$ era de esperar. Pero que

también sea $O(n)$, es fantástico. Debe ser por lo que se promocionan los de la segunda mitad cuando hundimos los de la primera, debe ser porque cada vez que ponemos un ítem en su lugar hundiéndolo, ponemos también algún otro en su lugar promocionándolo, o alguna cosa parecida, pero la cuestión es que tarda $\Theta(n)$. Lo normal, así a ojo, parece que tenga que ser $n/2\ log(n)$, o quizás $n/2\ log(n/2)$, o... no sé. En definitiva, podemos estar contentos. No demostraremos que sea $\Omega(n)$, porque está clarísimo. Con un bucle acotado por $n/2$, no se puede evitar pertenecer a $\Omega(n)$.

Para la demostración de que el tiempo de *crear_heap* es $O(n)$ suponemos que el árbol que nos representa el heap es completo (es decir, que $n \in \{3, 7, 15, ..., 2^{h-1}\}$). Entonces $h = log(n)$. En el nivel h, hay $n/2$ nodos. En el nivel $h-1$, $n/4$.

Imaginando el peor de los casos, calculemos el tiempo de cada nivel como si cada llamada a hundir tardara lo máximo posible.

Comencemos. Tenemos tan mala suerte, que hundimos los $n/4$ del penúltimo nivel, $h-1$ y para todos es necesario intercambiarlos. Hemos tardado $n/4$ veces $O(1)$. Vamos sumando. Una vez hundidos los $n/4$ nodos del penúltimo nivel, para el antepenúltimo, $h-2$, donde hay $n/8$ nodos, también tenemos la misma mala suerte, y hay que hundirlos todos hasta las hojas. Eso representa tener que hacer $n/8$ veces un trabajo que tarda $O(2)$. Ya llevamos $n/4\ O(1)\ +\ n/8\ O(2)$. Así iríamos haciendo hasta llegar a $h = 1$. En este nivel final, serían necesarias 2 llamadas a hundir como se puede suponer si os imagináis el árbol, o bien tomar el número $n/(n/2)$ que es el que toca.

Estas dos llamadas finales a hundir tardan $O(log(n-1))$ cada una. La suma total tendría la forma $n/4\ O(1)\ +\ n/8\ O(2)\ +\ \ldots\ +\ 2\ O(log(n-1))$. En lenguaje más preciso,

$$T(n) = \sum_{j=1}^{log(n-1)} n/2^{j+1} O(j) \tag{2.1}$$

$$= O\Big(n/2 \sum_{j=1}^{log(n-1)} j/2^j\Big) \tag{2.2}$$

Bien, está claro que para pasar de la expresión (2.1) a la (2.2), la única cosa que hemos hecho ha sido sacar del sumatorio todo lo que no depende de j, es decir, el término $n/2$.

Llegados a este punto, podemos sumar,

$$\sum_{j=1}^{log(n-1)} j/2^j = 1/2 + 2/4 + 3/8 + 4/16 + 5/32 + \cdots \leq 2.$$

De hecho, si sumáramos infinitos términos de esta serie, el valor límite, y por tanto el de la suma, sería 2. De todas formas, aquí ni siquiera esto. Sumamos tan sólo los $log(n-1)$ primeros términos, y por descontado que todo junto hace que para el algoritmo de crear el heap, $T(n) = O(n)$.

La clase *cola_de_prioridad_heap* vista en el Algoritmo 2.26 se muestra parcialmente en el Algoritmo 2.31. Se ha omitido todo lo que no concernía al heapsort, y se ha dejado la parte que el heapsort sí que utiliza. Como se puede observar, la implementación del procedimiento de crear el heap es sencilla. En la parte central del mismo Algoritmo 2.31 se ve que tan sólo hace uso de una de las dos operaciones internas de la clase. Se limita a hundir la primera mitad del heap en orden decreciente.

La función del constructor viene a ser una especie de envoltorio, que en un tiempo perteneciente a $\Theta(n)$ llena la memoria interna donde la clase guarda el vector. Esto ocupa la línea en la que se reserva la memoria y la siguiente, con el *for* de una sola línea. Una vez hay todos los elementos que han de constituir el heap en el vector, entonces se llama a *crear_heap()* que como se ha dicho actúa in situ.

Es interesante darse cuenta de que el orden de entrada de los valores en la estructura queda neutralizado por el grado de libertad que se ha dejado en el momento de su definición.

```
#include "item.h"

class cola_de_prioridad_heap
{
private:
    item *cola;
    int n;
    //...
    void hundir(int k, int n) {
        while (2*k <= n) {
            int j = 2*k;
            if (j<n && cola[j].prioridad > cola[j+1].prioridad) j++;
            if (cola[k].prioridad < cola[j].prioridad) break;
            item aux = cola[k]; cola[k] = cola[j]; cola[j] = aux;
            k = j;
        }
    }

    void crear_heap() {
        for (int i = n / 2; i > 0; --i) hundir(i, n);
    }

public:
    //...
    void heapsort(int*T, int N) {
        if (cola) delete [] cola;
        cola = new item[1+N];
        for (int n=1; n<=N; n++) cola[n].prioridad = T[n];
        crear_heap();
        item i;
        for (int j = n; j > 0; j--) {
            getmax(i);
            T[j] = i.prioridad;
        }
    }
};
```

Algoritmo 2.31 *Heapsort. Ordenación con colas de prioridad.*

Por lo que respecta a la eficiencia del heapsort como algoritmo de ordenación, observad que la segunda parte de la rutina requiere $O(n log(n))$, ya que se hacen n extracciones del máximo. Alguien podría argumentar que en cada extracción el tamaño del problema, n, se hace más chico, y por tanto, las n extracciones no son tales. Es más, la última tan sólo es $\Theta(1)$. Es cierto. De todas formas eso no nos permite reducir la cota superior que sigue siendo $O(n\ log(n))$.

En definitiva, el heapsort es $\Theta(n\ log(n))$, ya que como hemos visto es, por

una parte, $O(n\ log(n))$ y fácilmente podríamos encontrar una instancia que tardara $\Omega(n\ log(n))$.

¿Y por qué tanta historia?. ¿Por qué nos esforzamos tanto en demostrar que crear el heap tarda $\Theta(n)$ si después el algoritmo global de ordenación tarda igualmente $\Theta(n\ log(n))$?

La respuesta es sencilla. Hay veces que nuestro interés no se centra en tener todo el vector ordenado, sinó en tan sólo los cinco elementos máximos de un vector desordenado. O los diez números mayores. Entonces, fijaos que se podría resolver el problema con un tiempo perteneciente a $\Theta(n + 10\ log(n))$. Eso es el máximo entre n y $10\ log(n)$, que según cuando valga n, este máximo puede darse en el segundo candidato.

Dicho de otra manera. Gracias a que crear el heap tarda $O(n)$, para encontrar el k-ésimo elemento mayor de un vector desordenado de n elementos podemos tardar sólo $\Theta(k\ log(n))$ si eso es mayor que n y menor que $n\ log(n)$. Es muy interesante.

La pregunta que sale del corazón es... ¿A partir de qué valor de k, $1 \leq k < n$ nos interesa crear un heap en lugar de ordenar el vector? La respuesta es cuando $k\ log(n) > n$. Es decir, cuando $k > n/log(n)$. O sea, casi siempre.

2.4 Particiones

Como se ha introducido en la Sección 1.1.4, las clases de equivalencia son un concepto fundamental para el álgebra y el análisis matemático. La única manera categórica de realizar abstracciones conceptuales es haciendo uso de las particiones que definen las relaciones de equivalencia. Una relación es un predicado formulado sobre parejas de individuos de un universo, y para ser de equivalencia, además tiene que cumplir tres propiedades. La cosa buena es que cuando se define una relación de equivalencia sobre los individuos de una población, automáticamente estamos particionando la población en clases de equivalencia.

En síntesis, en esta sección se trata de ofrecer una estructura a la que le podamos introducir tantas parejas de individuos como deseemos. Se supone que cuando introducimos una pareja es porque satisface la relación de equivalencia por la cual estamos implementando la partición. Después, a la misma estructura, le queremos pedir si dos elementos son de la misma clase o no. Es decir, para cualquier elemento de la población, queremos que nos diga cuál es su representante de clase. Así, dos elementos serán de la misma clase cuando sus representantos coincidan.

Les particiones también son conocidas con el término inglés de *MF-sets*, acrónimo de *merge-find* sets. Les operaciones que caracterizan estas estructuras

de datos son, pues, aquéllas que hacen referencia a unir dos elementos, y a encontrar el representante de cualquier elemento.

Definición 2.9 Partición. *Estructura de datos que contiene elementos con claves, especializada en las operaciones de unir dos elementos, y en encontrar el representante de cualquier elemento.*

Una buena representación gráfica de como se implementarán las particiones sería un bosque, colección de árboles n-arios. Todos los individuos de una determinada clase son descendientes del representante de su clase. De cara a la implementación, como con las colas de prioridad, resolvemos las particiones en vectores semiestáticos, que sólo reserven memoria en el momento de su creación.

La dimensión del vector se corresponde con el número de elementos de la población que queremos segmentar en clases, n, y cada individuo vendrá representado por un índice del vector.

Tenemos así una eficiencia espacial de $\Omega(n)$ que está perfectamente bien ya que por definición, la clase a la que pertenece un individuo no es calculable a partir del individuo. Es decir, hay que guardar algún dato que etiquete cada individuo con la identificación de la clase a la que pertenece. El contenido del vector nos dirá la clase a la que pertenece cada individuo.

Establecemos la convención de que cuando el contenido del vector para un individuo de índice i es negativo, se trata de un representante de clase. Cada valor negativo en el vector representa la raíz de una nueva clase de equivalencia. Los individuos de índices con valor de contenido positivo pertenecen a la clase indicada en la posición indexada por este valor. Así pues, es una definición recursiva.

Para los valores negativos estamos disponiendo de un grado de liberdad semántica. Además de ser negativos (representantes de clase), podemos utilizarlos para alguna cosa. Está bien claro que el estadístico más interesante es el cardinal de la clase, eso nos servirá para mantener los árboles equilibrados.

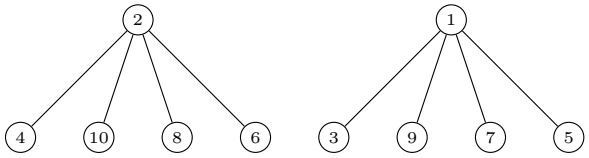
En definitiva, los valores contenidos en el vector que implementa la partición pueden ser de dos tipos.

Positivos: El índice del valor es de la clase indexada por el valor. Es una definición recursiva.

Negativos: El índice del valor es representante de clase. Además, en esa clase hay un número de elementos igual a este valor en positivo.

Pensemos en un universo formado por los números naturales del 1 al 10. Definimos sobre este conjunto la relación de equivalencia "tener la misma paridad". Entonces, deberíamos utilizar 8 llamadas a la operación *unir*. Uniríamos

el 1 y el 3, el 3, o el 1, y el 5, ... y luego, también el 2 y el 4, el 4, o el 2, y el 6, etcétera. Finalmente, la representación gráfica del bosque que obtendríamos, en ese caso formado por tan sólo dos árboles, se muestra en la Figura 2.26*(a)*.



(a)

-5	-5	1	2	1	2	1	2	1	2

(b)

Figura 2.26: *Representación gráfica de la estructura que implementa las particiones. (a) Arborescente; (b) Vectorial.*

Con la desordenación de los hijos de los árboles de la Figura 2.26*(a)* se pretende poner énfasis en que el orden entre los individuos de la población no tiene ni siquiera por qué estar definido. En la parte inferior, Figura 2.26*(b)* se nos muestra la representación vectorial de la partición. Por lo que se ha explicado más arriba, en la Figura 2.26*(b)*, los dos menos cincos significan que la clase con representante 1 tiene 5 elementos, y la del representante 2, también.

Un inconveniente que tiene la implementación de las particiones es que los árboles pueden degenerar. Hay que tener en cuenta que por lo que se ha dicho hasta ahora, la función recursiva que nos da el representante de clase de un individuo dado puede tardar $\Theta(n)$. Eso es poco eficiente.

Los dos árboles mostrados en la Figura 2.26*(a)* representan el mejor de los casos, que es cuando todos los individuos cuelgan directamente de sus representantes de clase.

En la implementación de la unión, sin embargo, la topología final del árbol depende del orden con el que se unan los elementos.

A fin de de evitar la degeneración, hay una decisión que se puede tomar de la manera más conveniente. Se trata del momento en que se hace una unión. Si, como se ha dicho, se dispone del cardinal de las clases a unir, puede hacerse que la que tenga más elementos quede como padre de la que tenga menos. Así, se podrá asegurar que la consulta del representante siempre sea $\Theta(log(n))$. Además, también es posible acelerar los procesos si cada vez que se consulta un representante se le actualiza el valor de su padre atándolo directamente a la

raíz. Así está implementado en el Algoritmo 2.32 donde se puede observar una versión sencilla de esta estructura de datos.

```
class particion {
public:

    int* t;
    int n;

    particion(int N) {
        n = N;
        t = new int[1+n];
        for (int i=1; i<=n; i++) t[i] = -1;
    }

    ~particion() {
        delete [] t;
    }

    int representante(int i) {
        return (t[i]<0)? i : t[i] = representante(t[i]);
    }

    void unir(int i, int j) {
        int ci = representante(i);
        int cj = representante(j);
        if (ci!=cj) {
            if (t[cj]>=t[ci]) {
                t[ci] += t[cj];
                t[cj] = ci;
            }
            else {
                t[cj] += t[ci];
                t[ci] = cj;
            }
        }
    }
};
```

Algoritmo 2.32 *Implementación de las clases de equivalencia con la clase* particion.

Un breve seguimiento. El constructor recibe el número de individuos de la población. Se crea un vector con este número de componentes, y se inicializan todos ellos a -1. En este momento, anterior a ninguna unión, si se pide por el representante de cualquier índice la respuesta es immediata. Se requiere $\Theta(1)$ para retornar el valor del mismo índice.

Supongamos que la primera llamada a unir elementos pide unir el 1 y el 2. Como en este momento ambas clases tienen tan sólo un elemento, hay un empate que se rompe con el uso del índice pasado como primer parámetro, es decir, en el ejemplo, el 1. Entonces el contenido del vector sería un -2 en la posición 1, y un 1 en la posición 2.

Ahora unimos el 3 y el 1, o sea, en el código del Algoritmo 2.32, $cj = -2$ y $ci = -1$. Iríamos por el *else*. Y saldríamos de la rutina con un -3 en la posición 1 y un 1 en la posición 3...

Respeto a la eficiencia de las operaciones, tenemos que gracias al equilibrio que forzamos al unir, las dos operaciones son $\Theta(log(n))$.

A lo largo de la primera sección dedicada a los diccionarios, se han ido mostrando implementaciones cada vez mejores para implementar las tablas de símbolos. Cuando la mejora no se ha reflejado en la eficiencia, sí que lo ha hecho en la capacidad. En síntesis, hemos comenzado con un diccionario no factible, implementado en un vector, que sólo aceptaba unas pocas entradas, y con las claves bien fijadas dentro de un margen muy pequeño. Además, el espacio dedicado al número de elementos que podía almacenar era muy limitado. De hecho, si se pudiera implementar un diccionario en un vector, sería una implementación óptima respeto a la eficiencia. Después, hemos probado de resolver los diccionarios con listas. Como mínimo hemos conseguido solucionar el problema de la capacidad, aunque la eficiencia era nefasta, de $\Theta(n)$ para cualquier operación. Un poco mejor, las tablas de dispersión. La primera, implementada con direccionamiento abierto, tenía el problema de la capacidad limitada, aunque conseguía aumentar la eficiencia de las listas con una constante multiplicativa. Probablemente no sea mejor el hashing con direccionamiento abierto que la implementación con listas, pero le hemos hecho un vistazo por cuestiones históricas. Las tablas de dispersión con encadenamiento separado, son claramente mejor que todo lo visto antes, ya que no tienen problemas de capacidad, y reducen la eficiencia de la implementación con listas en una constante multiplicativa. Una vez metidos en el uso intensivo de la memoria dinámica, vamos a parar a los árboles binarios de búsqueda, que tienen un peor caso como el de las listas, pero el caso medio y mejor superior a todo lo previo. Tan sólo tienen el problema del caso peor. Finalmente, con los árboles AVL, resolvemos cualquier problema que nos hayamos encontrado antes, tanto de capacidad, ahora cabe todo, como de eficiencia, que es $\Theta(log(n))$ para todas las operaciones. O sea que muy bien, hemos conseguido lo que nos proponíamos.

Después se han visto las colas de prioridad, estructuras especializadas en obtener el elemento más interesante de una colección. Tan sólo se ha descrito una implementación, los heaps. Les operaciones de promocionar y hundir que contienen colaboran en gran medida a sus finalidades.

El capítulo se ha cerrado presentando las particiones como estructuras de datos. Se ha visto una implementación sencilla, en un vector, y la implementación de las dos funciones características, *unir()* y *representante()*.

Capítulo 3

Dividir y Vencer

A lo largo de los años que he impartido esta materia, el equipo de profesores hemos ido probando diferentes ordenaciones de los mismos ocho temas que componen el temario. Siempre el primer tema ha sido el de la notación asintótica. Está bien claro que si durante todo el curso nos tenemos que dedicar a medir eficiencias, hay que empezar presentando el instrumento de medida. El segundo tema también ha sido siempre inalterable, estructuras de datos. Probablemente porque sea el tema más continuista con temas de materias anteriores. Y a partir de este punto hemos probado varias permutaciones de los temas. Cada ordenación diferente tenía sus razones. De todas ellas, me he quedado con ésta. Poner como tercer tema el de dividir y vencer me parece lo más correcto. Así, el curso queda ordenado, a grandes rasgos, por el mismo orden que las eficiencias de los algoritmos que se presentan. Otros semestres se ha dado grafos como tercer tema, pero a mi me parece que en éste que ahora abrimos, el de dividir y vencer, las eficiencias que se estudian todavía son todas polinómicas. Con los grafos abrimos la puerta a algoritmos de complejidad superior.

Este capítulo empieza con una introducción donde se expone la definición de los *Esquemas Algorítmicos*, ya que efectivamente, el de *Dividir y Vencer* es el primero que se menciona en este libro. Se sigue con el modelo o esquema algorítmico propiamente dicho, y se apuntan algunos comentarios para orientar al lector en el momento de comprender la estrategia de dividir y vencer. Después, como ejemplos, es muestran los dos últimos algoritmos de ordenación que se presentan en el libro. La ordenación rápida o de Hoare, que aquí llamaremos *quicksort*, y la ordenación por fusión, o *mergesort*. Es notable el hecho de que a pesar de no tener un capítulo dedicado a los algoritmos de ordenación, gran parte de los existentes son iluminados a lo largo de los diferentes temas a modo de ejemplo.

El capítulo cierra con algunos algoritmos que ilustran la máxima expresión de esta técnica, y que le dieron nombre. Al lector estudiante, se le aconseja que no se preocupe por el hecho de que los ejemplos de este capítulo resulten

de una eficiencia sorprendente. Puede resultar duro observar como el algoritmo para multiplicar que aprendimos en la escuela puede ser mejorado con la observación minuciosa del proceso. Y por esta razón, vaya por delante, que los ejercicios académicos que nos podemos encontrar sobre el tema de dividir y vencer no requieren ninguna astucia elitista. Normalmente serán variaciones nada complicadas de los algoritmos que se presentan en este capítulo.

3.1 Introducción

En esta introducción se muestra el algoritmo más característico del esquema algorítmico de dividir y vencer. Se insiste en las versiones recursiva e iterativa, poniendo énfasis en que un esquema algorítmico no está comprometido con ninguna de las dos implementaciones a pesar de que el modelo o plantilla que representa el esquema se muestre en una versión recursiva.

3.1.1 Esquemas Algorítmicos

En las últimas décadas ha habido una cierta obstinación en la formalización del conocimiento algorítmico, tal como debe ser. Uno de sus frutos es el establecimiento de lo que se ha dado en llamar esquemas algorítmicos. De momento, se consideran cuatro de ellos:

- Esquema Algorítmico de Dividir y Vencer *(Divide and Conquer)*: A partir de la instancia inicial, fragmentan sucesivamente la instancia actual de tamaño n en k instancias más chicas, hasta conseguir instancias tan pequeñas que se pueden solucionar trivialmente. Después, habiendo solucionado los *subproblemas* triviales, combinan sucesivamente las soluciones obtenidas para obtener soluciones a los problemas que habían aparecido en la fragmentación, y también combinan las soluciones de éstos para obtener la solución del problema inicial. A grandes rasgos, es el esquema algorítmico que proporciona algoritmos más eficientes.

Los otros tres esquemas se orientan a resolver problemas de optimización, es decir, de maximización o minimización. Se verá en los próximos capítulos que en estos problemas se trata de tomar un conjunto de decisiones. Son decisiones poco o muy relacionadas entre ellas. En el supuesto de que se trate de decisiones binarias, entonces estos problemas se podrían resolver considerando todas las combinaciones en una tabla de verdad, por el cuento de la vieja. Es decir, probando todas las posibilidades que hay entre las n decisiones. Tan solo se debería evaluar la función a minimizar para cada entrada de la tabla, y escoger el óptimo. Claro, tener en cuenta todas las posibles combinaciones entre las n decisiones, hace que el problema se nos escape de las manos, y los tiempos

trepan hasta $\Omega(2^n)$ o más. Se trata de encontrar algoritmos polinómicos, o sea, con el tiempo por debajo de $O(n^k)$ para alguna $k \in \mathbb{R}$, que nos den el óptimo siempre que pueda ser. Los tres esquemas algorítmicos que se focalizan en este tipo de problemas son,

- Esquema Algorítmico de Algoritmos Voraces *(Greedy Algorithms)*: En este tipo de algoritmos, se prioriza el hecho de acabar pronto más que el hecho de tener una solución óptima. El esquema voraz, por encima de todo, toma una decisión en cada iteración de un bucle. Se ve en detalle en el Capítulo 5.

- Esquema Algorítmico de Programación Dinámica *(Dynamic Programming)*: Es una estrategia con el espíritu de dividir y vencer, en cuanto a la fragmentación en subproblemas. Tanto el problema como los subproblemas se solucionan moviéndose por un camino de subsoluciones óptimas que, en cada paso, aumenta el tamaño del subproblema hasta llegar al tamaño de la instancia original. Es un esquema de naturaleza iterativa, pero en cambio, rellena estructuras de datos dinámicas con reglas que obedecen a recurrencias. Es el esquema algorítmico que más se soporta en el uso de la memoria dinámica. Acostumbra a dar eficiencias temporales lineales, o polinómicas como mucho, pero en cambio las eficiencias espaciales no son tan satisfactorias. Este esquema se analiza en el Capítulo 6.

- Esquema Algorítmico de Búsqueda Exhaustiva *(Branch and Bound)*: Es un esquema de filosofía inversa, en cierta manera, a la de los algoritmos voraces. Buscan la solución al problema de optimización en base a comparar los resultados de todas las posibles combinaciones de decisiones que se puedan tomar. Por esta razón, también se le conoce como esquema enumerativo. Pueden tardar tanto en resolver el problema, que resulte que no valga la pena buscar la solución por esta vía. Sin embargo, siempre que dan una solución, podemos estar seguros que es una solución óptima. Se ven en detalle en el Capítulo 7.

3.1.2 Recursividad

Que se asume al lector ciertos conocimientos de recursividad está claro. Desde el primer capítulo hemos estado hablando con fluidez de ella. Aquí, pues, bajo este título se pretende introducir el tema de dividir y vencer. Como ya se ha dicho allá, una función recursiva es aquélla que en algún caso se llama a sí misma.

Clasificamos todos los algoritmos recursivos en dos grupos, según el tipo de recursividad que utilicen.

- *Recursividad de ida* es aquélla en la que cuando se resuelven los casos triviales, el problema ya está resuelto.

- *Recursividad de vuelta* es aquélla otra en la cual cuando el árbol de llamadas recursivas se repliega, va acumulando los resultados de los subproblemas pequeños, de manera que se consigue la solución cuando se ha vuelto de todas las llamadas.

Aunque no necesariamente la técnica de dividir y vencer pase por algoritmos recursivos, es conveniente asociar los dos conceptos.

La búsqueda dicotómica es el algoritmo emblemático del esquema algorítmico de dividir y vencer.

En el Algoritmo 3.1 se puede ver una implementación recursiva de la búsqueda dicotómica, y en el Algoritmo 3.2 un procedimiento equivalente implementado iterativamente.

Este algoritmo sirve para encontrar un valor x en una secuencia ordenada de valores. Si la secuencia no fuera ordenada no podríamos utilizar este algoritmo de búsqueda. Y en definitiva, la razón por la cual se ordenan las secuencias es precisamente ésta. Acelerar las búsquedas.

Que quede claro, pues, este extremo. Y ahora agarraos, un absurdo frecuente que cometen los estudiantes, y que en definitiva debería conducir a cualquiera al suspenso absoluto de la materia completa, es implementar búsquedas dicotómicas en vectores no ordenados!

```
int busqueda_dicotomica(double T[ ], int i, int d, double x)
{
    if (i<d) {
        int m = (i+d)/2;
        if (x>T[m]) return busqueda_dicotomica(T,m+1,d,x);
        if (x<T[m]) return busqueda_dicotomica(T,i,m-1,x);
    }
    if (x==T[i]) return i;
    else return -1;
}
```

Algoritmo 3.1 *Implementación recursiva de la búsqueda dicotómica.*

Para calcular la eficiencia en los Algoritmos 3.1 o 3.2 es necesario en primer lugar definir n, ya que no aparece explícitamente en el código. Tomaremos lo que sabemos que es el tamaño de la instancia, $n = d - i + 1$. La condición $i < d$ podría anotarse como $n > 1$.

Como ya se había visto en capítulos anteriores, el algoritmo de búsqueda dicotómica pertenece a $\Theta(log(n))$. Eso se puede comprobar fácilmente utilizando

el Teorema Maestro para las recursividades divisoras. Por esta vía, ya se ha visto que $a = 1$, $b = 2$, y $k = 0$. Entonces nos encontramos en el caso $a = b^k$ y por tanto la resolución de la recurrencia nos da un tiempo en $\Theta(n^k log(n))$.

Para el caso del Algoritmo 3.2, hay que contar cuántas veces se ejecutará el bucle en función de n. En definitiva, el tamaño $n = d - i + 1$ se verá reducido a la mitad en cada iteración. El número de iteraciones total será pues $\Theta(log(n))$.

```
int busqueda_dicotomica(double T[ ], int i, int d, double x)
{
    while (i<d) {
        int m = (i+d)/2;
        if (x>T[m]) i = m;
        if (x<T[m]) d = m;
    }
    if (x==T[i]) return i;
    else return -1;
}
```

Algoritmo 3.2 *Implementación iterativa de la búsqueda dicotómica.*

Como ya se había dicho, la eficiencia de un algoritmo no depende de si la implementación es recursiva o iterativa.

3.1.3 Ineficiencia

Hay cierto peligro en el uso de la técnica de dividir y vencer. Se requiere sentido común para darse cuenta de que los subproblemas deben de ser *independientes*. Con este término se pretende hacer referencia a que no se repitan cálculos para distintos subproblemas. Un caso flagrante, en el cual la elegancia del algoritmo puede deslumbrar, es el caso de la secuencia de Fibonnaci. En la Sección 1.6.3 ya se ha visto en detalle este caso. El Algoritmo 3.3 reproduce el código de la función.

```
int fibonacci(int n)
{
    if (n<=2) return 1;
    return fibonacci(n-1) + fibonacci(n-2);
}
```

Algoritmo 3.3 *Cálculo de los números de Fibonacci.*

El aspecto de este algoritmo podría hacer pensar que es una buena implementación. Utiliza la técnica de dividir y vencer. No obstante, ya se ha mostrado en la Sección 1.6.3 con el Teorema Maestro I que es un algoritmo $O(2^n)$ y por tanto inadmisible para números grandes. Y sin el teorema, haciendo caso al sentido común, también se ve que con la función del Algoritmo 3.3, *fibonacci(*$n-2$*)* se calculará dos veces. Pero es que *fibonacci(*$n-3$*)* ya es calculará cuatro veces, y el siguiente, ocho. Así pues, el error está en que los subproblemas no son independientes. Para estos problemas hay que utilizar la programación dinámica y no la técnica de dividir y vencer.

3.2 Esquema Algorítmico de Dividir y Vencer

En el Esquema 3.1 tenemos una especie de plantilla que viene a ser el model de los algoritmos que utilizan la técnica de dividir y vencer. Este esquema es recursivo. La idea básica radica en la descomposición y la combinación.

```
algoritmo dividir_y_vencer(problema)
{
    si trivial(problema) retorna solucion(problema)
    p_1, p_2, ..., p_k = descomponer(problema)
    para i=1 hasta k hacer
        s_i ← dividir_y_vencer(p_i)
    fpara
    retorna combinar_soluciones(s_1, s_2, ..., s_k)
}
```

Esquema 3.1 *Esquema Algorítmico de Dividir y Vencer.*

Este modelo tiene que ser concebido como una referencia poco estricta. Es decir, encontraremos casos donde la descomposición es trivial, o la combinación de soluciones inexistente.

Puntualizaciones sobre el Esquema 3.1.

- No aparece el tamaño de la instancia, n. Va implícita en el objeto *problema*.
- La función booleana *trivial*, más que ser función del problema, tiene que ver exclusivamente con el tamaño del problema. Cuando n sea más pequeña que una cierta constante retornará cierto, y si es mayor, falso.
- La función *solucion* ha de ser eficiente, $\Theta(1)$. No es de extrañar, ya que siempre se llamará cuando el tamaño del problema sea lo bastante pequeño en términos absolutos.

- La función *descomponer*, en recursividades de ida, desempeña la tarea más importante por la cual se implementa el algoritmo.

- El parámetro k es una constante pequeña. Dos, cuatro, ocho... y gracias a esto en los algoritmos reales que utilizan la técnica de dividir y vencer no aparece el bucle que sí que aparece en el esquema algorítmico. En ningún caso k dependerá de la instancia. Por otra parte caracterizará la eficiencia global. Normalmente irá asociada al algoritmo en cuestión. Para el caso de la búsqueda dicotómica, por ejemplo, $k = 2$.

- La función *combinar_soluciones*, en recursividades de vuelta, desempeña la tarea más importante por la cual se implementa el algoritmo.

3.3 Ordenación Rápida *quicksort*


Anthony Hoare

Charles Antony Richard Hoare (Sri Lanka, 1934-...) es un ingeniero en computación especializado en lógica. Es uno de los grandes dinosaurios de la programación informática. Ideó la lógica de Hoare que tenía el ambicioso propósito de establecer la corrección de los programas informáticos a partir de sistemas de precondiciones, con el rigor de la lógica matemática. También es obra suya uno de los libros más citados sobre descripciones de patrones de interacción entre procesos, donde se establece un lenguaje, el CSP, para la comunicación de procesos secuenciales. Además, se ha dedicado a la traducción entre lenguajes hombre máquina. Y en ese ámbito, ideó el método de ordenación rápida. Éste es el algoritmo más rápido conocido para ordenar, y el más ampliamente utilizado del mundo hoy día.

El Algoritmo de ordenación rápida utiliza recursividad de ida. La idea básica consiste en tomar cualquier elemento de la secuencia a ordenar, y distribuir todos los demás en dos grupos. Los menores, y los mayores que el elemento escogido, que es cualquiera, y que se le llama *pivote*.

Utilizaremos intensamente el intercambio de valores entre variables. El código que se muestra en este libro pretende utilizar las mínimas librerías posibles del lenguaje C. Es por esta razón que aunque quizás resulte trivial, en el Algoritmo 3.4 se muestra un procedimiento que en esta sección será de gran utilidad.

```
void swap(double& a, double& b)
{
    double aux = a;
    a = b;
    b = aux;
}
```

Algoritmo 3.4 *Rutina* swap *para el intercambio de valores entre dos variables.*

Por cierto, ya que estamos hablando entre expertos en programación, que se sepa que para hacer un intercambio entre valores de dos variables no es estrictamente necesario el uso de una variable adicional. Un código como el mostrado en el Algoritmo 3.5 hace un intercambio de los valores de las variables a y b sin necesidad de ninguna variable auxiliar, aunque con un cierto riesgo de desbordamiento. En cualquier caso, somos humanos, y a menudo conviene implementar operaciones de la manera más legible. En particular, para las operaciones que son $\Theta(1)$, conviene que prevalga la legibilidad. Por esto, no conviene utilizar funciones como la del Algoritmo 3.5.

```
void swap(double& a, double& b)
{
    a = a + b;
    b = a - b;
    a = a - b;
}
```

Algoritmo 3.5 *Intercambio de valores de variables sin utilizar espacio auxiliar.*

Bien, volvamos al *quicksort* comenzando por el principio.

3.3.1 Esencia

Tenemos un conjunto de valores para ordenar en un vector. Tomamos el primer valor, por ejemplo, de referencia, el *pivote*. Utilizaremos dos índices que servirán para poner los demás valores donde les corresponda respeto al pivote. Uno empezará desde el final e irá decrementando hasta encontrar algún valor menor que el pivote. El otro, empezando desde el principio, irá incrementando hasta encontrar un valor mayor. Acabamos cuando los dos índices se encuentren. Eso puede suponer varias iteraciones, y en cada una se hace algún intercambio favorable a la ordenación. Cuando se crucen, entonces en aquella posición queda el pivote. Es lógico, mientras no se encuentren, significa que quedan elementos

desordenados respeto al pivote, y por tanto, los vamos intercambiando. Tradicionalmente, a esta rutina se le ha llamado partición, cosa que no tiene nada que ver con la Sección 2.4. Aquí se refiere a partir en dos. Allí, a una clasificación en grupos.

```
int particion(double T[], int i, int d)
{
    int p = i;
    while (1) {
        while (p < d && T[p]<=T[d]) --d;
        if (p==d) return p;
        swap(T[p],T[d]);
        p = d;

        while (i < p && T[i]<=T[p]) i++;
        if (i==p) return p;
        swap(T[i],T[p]);
        p = i;
    }
}
```

Algoritmo 3.6 *Esencia del* quicksort.

En el Algoritmo 3.6 se muestra una posible implementación de lo que, en última instancia, hace la tarea de ordenar en el algoritmo de ordenación rápida.

Con un procedimiento como el del Algoritmo 3.6 obtenemos tres cosas favorables para la ordenación.

- Un elemento, el pivote, en su posición final, p, que es la que se retorna.
- Un segmento del vector, $[e, p-1]$, con valores menores que el pivote.
- Otro segmento del vector, $[p+1, d]$, con valores mayores que el pivote.

Así pues, llamando recursivamente al algoritmo de ordenación con las partes inferior y superior del pivote, ordenaremos el vector.

Filosóficamente, tiene interés el hecho de que las comparaciones se rentabilizan un orden de magnitud más que en el caso de ordenaciones simples como la de inserción o de selección. Es decir, se extrae más información de las comparaciones. En el algoritmo de ordenación por selección, por ejemplo, cada vez que hacemos una comparación, no nos importa nada de nada la que hemos hecho previamente. Aquí, sí. Este procedimiento es más sabio porque sabe cosas como que *de aquí hasta al final, todos son más pequeños que éste.*

La eficiencia del procedimiento *particion* es $\Theta(n)$, ya que entre los dos bucles *while* interiores al bucle principal se recorrerá el vector completo una sola vez. El número de iteraciones del *while* principal depende del contenido del vector, pero en cualquier caso, el tiempo de ejecución de la función completa será siempre proporcional al tamaño, $n = d - i + 1$, del vector T.

3.3.2 Algoritmo

A partir de la observación del Algoritmo 3.6, se ve que el quicksort no requiere ningún espacio auxiliar, igual que el heapsort. La ordenación se hace en el mismo espacio de memoria en el que inicialmente teníamos el vector desordenado. De hecho, esta forma de ordenar hace un uso intensivo del acceso directo a los elementos. Y por esta razón, el único espacio adicional necesario es el elemento *aux* para hacer los intercambios.

La estructura de dividir y vencer que realiza la ordenación de un vector hace uso de la acción descrita en el Algoritmo 3.6, tal como se muestra en el Algoritmo 3.7. El procedimiento recibe de entrada un vector y los índices izquierdo y derecho del segmento que se pretende ordenar. O sea, que para ordenar el vector completo se usará una llamada del tipo

quicksort(T,0,n-1).

Es un algoritmo recursivo que en el caso trivial no hace nada. El hecho de que el trabajo se haga antes de las llamadas nos dice que la recursividad es de ida.

```
void quicksort(double T[], int i, int d)
{
    if (i<d) {
        int p = particion(T,i,d);
        quicksort(T,i,p-1);
        quicksort(T,p+1,d);
    }
}
```

Algoritmo 3.7 *Ordenación rápida.*

En la Figura 3.1 se puede contemplar la evolución de los índices dentro de la función *particion*. Para el caso del ejemplo, el vector inicial tendría el valor $T = \{3, 1, 5, 2, 4\}$.

Todo lo que se muestra en la Figura 3.1 es sólo la primera llamada a la función *particion*. Las líneas continuas significan iteraciones principales. Las

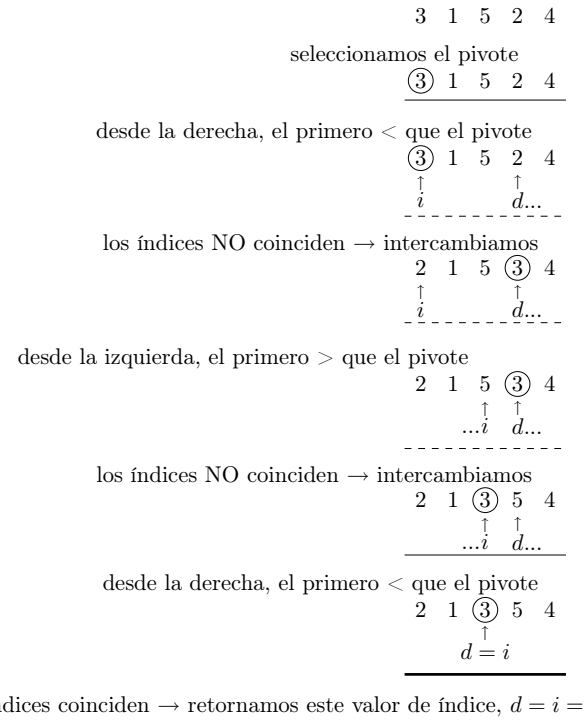


Figura 3.1: *Primera llamada a* particion *con* $T = \{3, 1, 5, 2, 4\}$.

discontinuas significan cada uno de los bucles más interiores. Tal como ya se ha mencionado, la función nos retorna el índice donde finalmente queda el pivote. Para el caso del ejemplo, el 2, que es el correspondiente al valor $T[2] = 3$ (aquí trabajamos con vectores comenzando en el índice cero). O sea, el vector en la salida de la primera llamada a la función *particion* nos queda $T = \{2, 1, 3, 5, 4\}$. Es fácil verificar que las tres cosas favorables es cumplen. Para seguir la ejecución completa, tan solo quedan las dos llamadas correspondientes a la primera parte del vector, $T = \{2, 1\}$ y a la segunda $T = \{5, 4\}$.

Tenemos dos cajas, una con tornillos y la otra con roscas. Todos los tornillos son de diferentes tamaños. Todas las roscas son de diferentes tamaños. Cada tornillo encaja únicamente con una rosca. Si probamos a encajar un tornillo cualquiera con una rosca cualquiera, podemos obtener tres resultados,

- El tornillo es más pequeño que la rosca.
- El tornillo encaja con la rosca.
- El tornillo es más grande que la rosca.

O sea, que si dos tornillos son más grandes que una misma rosca no tenemos ninguna información de cuál de los dos tornillos es más grande.

El problema consiste en aparejar cada tornillo con su rosca. O sea, en ordenarlo todo.

La mejor manera de atacar el problema sería cogiendo un tornillo cualquiera y probando todas las roscas de la caja con ese mismo tornillo. Si el tornillo no entra en la rosca, dejamos la rosca a la izquierda, donde dejaremos las chicas. Si encajan la dejamos en medio, y si la rosca es más grande que el tornillo, la dejamos a la derecha. Cuando hayamos probado todas las roscas con ese tornillo, tenemos las más pequeñas a un lado. También, el tornillo con su rosca en el medio, y las más grandes al otro lado. Entonces cogemos otro tornillo cualquiera de la caja, lo comparamos con la rosca del centro, que encajaba con el tornillo anterior, y si es más grande, repetimos todo el proceso con el montón de roscas del lado de las grandes. Y si es más pequeño, con las pequeñas. De esta manera trabaja la ordenación rápida.

3.3.3 Eficiencia

Para analizar la eficiencia de un algoritmo recursivo, otra vez recurrimos a los Teoremas Maestros. Para esto nos hace falta saber cuántas llamadas recursivas hace el algoritmo en tiempo de ejecución. Són dos, $a = 2$. Por otro lado, como ya se ha visto, $T_{particion} = \Theta(n)$, y por tanto $k = 1$.

Respeto al tercer parámetro..., hay casos. El caso mejor no es fácilmente identificable, pero el caso medio es lo bastante amplio como para incluirlo. Consideremos equivalentes los casos medio y mejor.

Si tenemos suerte, el valor que cojamos como pivote caerá en medio del intervalo de todos los valores. Entonces la variable p retornada por la función *particion* del Algoritmo 3.7 tomaría el valor medio entre i y d. En este caso, dividiríamos el problema en dos y por tanto podríamos utilizar el Teorema Maestro II para las recursividades divisoras con el parámetro $b = 2$.

En cambio, si tenemos mala suerte, el valor que cojamos como pivote puede ser el valor mínimo, o el máximo de los valores del vector, entonces la variable con el índice retornado por la función de partición valdrá i, o d. Deberíamos utilizar el Teorema Maestro I para las recursividades substractoras con el parámetro $c = 1$.

Finalmente, para concluir el análisis, razonablemente asumimos que el valor del pivote que se coge, en el caso medio, es el valor medio de los valores del vector a ordenar. O sea,

$$T_{quicksort}(n) = \begin{cases} \Theta(n\ log(n)) & \textit{en el caso medio (y en el mejor),} \\ \Theta(n^2) & \textit{en el caso peor.} \end{cases}$$

Es decir, $\Theta(n\ log(n))$ cuando los valores estén distribuidos aleatoriamente, y $\Theta(n^2)$ si el vector de entrada ya estaba ordenado.

Fijaos bien que el algoritmo de ordenación rápida, aún teniendo el peor caso peor de los algoritmos de ordenación, es el más utilizado. En la vida real el caso peor es produce lo bastante excepcionalmente como para asumir el coste que representa poderse encontrar con él. Es en el caso medio donde la constante oculta del quicksort es la más reducida, y por esta razón, este es el algoritmo más rápido.

3.3.4 Variantes

Específicamente, las mejoras que se han ido confeccionando para el algoritmo de ordenación rápida se focalizan en la selección del pivote. Para asegurarnos que no ocurrirá el caso peor, podemos escoger el valor del pivote aleatoriamente. A fin de de no modificar el Algoritmo 3.7, tan solo sería necesario hacer un intercambio de valores, entre el de un índice cualquiera y el valor de la izquierda del vector, antes de llamar a la función que particiona.

En general, para los algoritmos de ordenación recursivos, hay una forma de hacer que sean más rápidos, a nivel de constante oculta. Esta forma no tiene más secreto que dejar de ordenar a partir de un tamaño suficientemente pequeño.

Así somos los humanos por naturaleza. Pensad en vuestro piso. ¿Está ordenado? Hombre, casi todo el mundo tiene los fogones en la cocina, y la cama en la habitación. A grandes rasgos está ordenada. En cambio, también es verdad que casi todo el mundo tiene cajones en casa con muchas cosas sin ningún orden. Cuando el espacio es lo bastante reducido, no nos hace falta más ordenación. Nadie establece un orden hasta el punto de saber que en un cajón hay un bolígrafo en la derecha y unas gafas en la izquierda. Dentro del cajón, cuando buscamos alguna cosa, ya usaremos otro tipo de algoritmo de búsqueda, como el secuencial.

En el ámbito de los archivos, tampoco es muy grave saber que si buscamos una ficha en un fichero ordenado alfabéticamente y no la encontramos deberemos buscar tres fichas anteriores o posteriores a partir del lugar donde debería estar. A veces, resulta conveniente dejar las cosas desordenadas dentro de unos límites.

3.3.5 Selección Rápida *quickselect*

Una aplicación adicional de la ordenación rápida es la selección rápida.

Está claro, que si de un vector no ordenado nos interesa el mínimo valor haremos una búsqueda secuencial. Eso será así cuando sólo nos interesa el mínimo.

Si nos interesase el segundo valor menor, el algoritmo que utilizaríamos ya no está tan claro. Probablemente haríamos una primera búsqueda secuencial para obtener el mínimo, y después otra búsqueda para obtener el mínimo diferente del obtenido en la primera búsqueda. Ya se ve, que si nos interesara el octavo o noveno más pequeño, este algoritmo ya no serviría. Llegados a un cierto punto, valdría más la pena ordenar el vector del todo, y entonces ya tendríamos el más pequeño, el segundo más pequeño, el tercero, y así hasta el último que sería el máximo.

Así pues, para entendernos, formalicemos el problema de selección.

Definición 3.1 Problema de Selección. *El problema de selección consiste en, dado un conjunto de n valores indexables pero no ordenados, encontrar el k-ésimo más pequeño, siendo* $k \leq n$.

No es lo mismo que el caso del heapsort, explicado como utilidad adicional de las colas de prioridad. Allí, se trataba de saber los k elementos mínimos de un vector desordenado. Aquí sólo se quiere saber el k-ésimo si estuvieran ordenados. Con el heapsort se obtiene más información a un precio de $\Theta(k\ log(n))$. Con el quickselect, menos información a un precio de $\Theta(n)$ como se ve seguidamente.

No parece demasiado difícil deducir cómo resolveremos este problema con una ligera modificación del Algoritmo 3.7. Sólo hay que recordar que después de cada llamada a particionar obtenemos en p el índice que le corresponde al pivote si el vector estuviera ordenado. Por tanto, si el parámetro de entrada al algoritmo, k, fuera menor que el índice p continuaríamos haciendo la ordenación por la parte izquierda del vector. Si tuviéramos la suerte de que $k = p$, entonces ya habríamos acabado. Y si $k > p$, entonces continuaríamos por la derecha.

```
double quickselect(double T[], int i, int d, int k)
{
        if (i<d) {
                int p = particion(T,i,d);
                if (k<p) return quickselect(T,i,p-1,k);
                if (k==p) return T[p];
                if (k>p) return quickselect(T,p+1,d,k);
        }
}
```

Algoritmo 3.8 *Selección rápida del k-ésimo valor menor en un vector desordenado.*

En el Algoritmo 3.8 se muestra una posible implementación para encontrar el k-ésimo elemento menor de un vector desordenado en tiempo... reflexionemos:

$a = 1$, $b = 2$ en el caso medio, $k = 1...$ o sea $a < b^k$ que es $\Theta(n)$. Si no lo veis claro, deberíais repasar la Sección 1.6.4.

3.4 Ordenación por Fusión *mergesort*


John von Newmann

John von Newmann (1903-1957) fue un ingeniero químico nacido a Budapest que desde muy chico había demostrado gran habilidad con las matemáticas entre otras disciplinas. En la segunda guerra mundial ya estaba nacionalizado norteamericano y participó ayudando al ejército aliado. Entonces construyó, con otros, el ordenador considerado como el iniciador de la informática, ENIAC. Se especializó en arquitectura de computadores basados en máquinas de Turing. Por *arquitectura von Newmann* se describe aquel tipo de computadores que comparten la memoria para almacenar indiferentemente datos y programas. O sea, los normales. A Newmann también se li atribuye este algoritmo de ordenación, uno de los más antiguos.

El Algoritmo de ordenación por fusión utiliza recursividad de vuelta. La idea básica consiste en considerar ordenadas las secuencias de un solo elemento. Así, dividiremos y dividiremos el vector a ordenar, hasta que tengamos secuencias de un solo elemento. Y entonces, de vuelta, iremos recogiendo secuencias ordenadas cada vez mayores. De hecho, cada vez serán el doble de grandes. Si lo consultáis en la wikipédia podréis ver una animación muy curiosa de una ordenación de puntos en el plano.

3.4.1 Esencia

Los problemas que se solucionan recorriendo el espacio de salida con variables independientes, y para cada posición de este espacio calculan qué valor ha de contener, son conocidos como *problemas inversos*. Como por ejemplo la ordenación por inserción. En general, este tipo de problemas se caracterizan por tratar los datos de entrada como si fueran un almacén del cual extraen la información necesaria para rellenar el valor de la salida. Ejemplos de estos problemas son el producto de matrices, o la geocorrección de imágenes cartográficas. En el caso del producto de matrices, cada componente de la matriz resultante es calculado a partir de las dos matrices de entrada. Para el caso de la geocorrección, quizás merezca la pena introducir que geocorregir una imagen significa obtener un mapa de una zona geográfica concreta, definida por ejemplo en coordenadas longitud latitud, a partir de fotografías satélite que se sobreponen haciendo un recubrimiento de la zona. Para cada posición del territorio, el programa que realiza la geocorrección calcula de qué fotografías, y concretamente de qué píxe-

les en su interior, debe obtener el valor del color en ese punto longitud latitud. También, el mergesort se plantea la ordenación como un problema inverso.

Supongamos que tenemos dos secuencias ordenadas. La fusión de las dos será otra secuencia ordenada de longitud igual a la suma de las dos primeras. El procedimiento *merge* mostrado en el Algoritmo 3.9 realiza precisamente esta tarea.

```
void merge(double c[], int l, double a[], int m, double b[])
{
    int i = 0;
    int j = 0;
    for (int n=0; n<l+m; n++) {
        if (i==l) {c[n] = b[j++]; continue;}
        if (j==m) {c[n] = a[i++]; continue;}
        c[n] = (a[i] < b[j]) ? a[i++]:b[j++];
    }
}
```

Algoritmo 3.9 *Esencia del mergesort.*

La instrucción del C estándar *continue* dentro de un bucle *for*, salta directamente al final del bucle y continua con el siguiente valor de índice del *for*. Podrían ser alternativas completas utilizando *else*'s. Se presenta de esta manera para tener un aspecto más fiel a la naturaleza commutativa de la operación.

En el Algoritmo 3.9 hay dos secuencias de entrada, a de longitud l, y b, de longitud m. Y una secuencia de salida, c que finalmente tendrá longitud $n = l + m$. Es de lo más sencillo. En cada iteración del bucle se habrá colocado un elemento más en la secuencia c de salida.

Dentro del bucle, las dos primeras líneas sirven para cuando alguna secuencia de entrada se ha agotado. Entonces, la única cosa que resta por hacer es copiar los elementos de la otra. En la última línea del bucle, se trata el caso normal. Es la esencia de la esencia. Mientras queden elementos en las dos entradas, se selecciona siempre el menor de los dos para colocar en la salida, y se incrementa el índice en la secuencia de la que se haya extraído el elemento.

Con todo, recuerda un poco el funcionamiento de una cremallera, con la diferencia que en la salida no necesariamente quedarán los elementos alternados, sinó que pueden pasar dos elementos de un mismo lado seguidos, o tres, o más, sin que pase ninguno del otro lado.

La eficiencia del Algoritmo 3.9 corre con n. Eso es $T_{merge} = \Theta(n)$, llamando n a la longitud de la secuencia resultante.

3.4.2 Algoritmo

El procedimiento de dividir y vencer que realiza la ordenación por fusión de un vector hace uso de la acción descrita en el Algoritmo 3.9, tal como se muestra en el Algoritmo 3.10. O sea, sólo utiliza el procedimiento *merge* con $l \simeq m$, o sea similares. Este algoritmo recibe de entrada un vector y los índices izquierdo y derecho del segmento a ordenar. Para ordenar el vector completo será necesario llamar a

mergesort(T,0,n-1).

Es un algoritmo recursivo que en el caso trivial no hace nada. El hecho de que la tarea esencial se realice después de las llamadas hace que la recursividad sea de vuelta.

```
void mergesort(double T[], int i, int d)
{
    if (i<d) {
        int m = (i+d)/2;
        mergesort(T,i,m);
        mergesort(T,m+1,d);
        merge(T,i,m,d);
    }
}
```

Algoritmo 3.10 *Ordenación por fusión.*

Se empieza fragmentando el segmento a ordenar tantas veces como haga falta hasta tener subsegmentos de una sola componente, que los consideramos ordenados. A partir de ese extremo que es el caso trivial, retornamos aparejando parejas, y parejas de parejas.

La llamada a la función *merge* del Algoritmo 3.10 no respeta el paso de parámetros mostrado en el Algoritmo 3.9. En el Algoritmo 3.11 se muestra la implementación de una función intermedia. En ella se recibe el vector de reales T, y tres índices (izquierdo, medio, y derecho) entre los que se realizará la ordenación. Entonces parte el vector a ordenar T en dos vectores. Uno es a, de dimensión n con la primera mitad de T. El otro es b, de dimension m, con la mitad derecha de T. Luego declara un espacio auxiliar c de dimensión $n + m$. Llama a la función *merge*, y finalmente copia el resultado del espacio auxiliar c en T.

```
void merge(double T[], int i, int mitad, int d)
{
    int n = mitad-i+1;
    double* a = &T[i];
    int m = d-(mitad+1) + 1;
    double* b = &T[mitad+1];
    double* c = new double[n+m];
    merge(c,n,a,m,b);
    for (int ii=i; ii<=d; ii++) T[ii] = c[ii-i];
    delete [] c;
}
```

Algoritmo 3.11 *Transformación de parámetros para la ordenación por fusión.*

En la Figura 3.2 se puede ver que sólo se llama a la función *merge* después de haber llamado a *mergesort* con secuencias de un solo elemento.

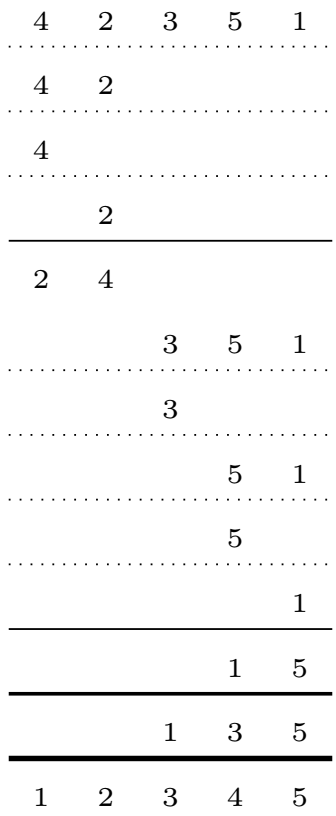


Figura 3.2: *Secuencia de llamadas a la función* mergesort *para ordenar un vector de 5 elementos. Las líneas continuas indican ejecución de la función* merge.

Las líneas punteadas significan llamadas recursivas sucesivas. Las líneas con-

tinuas significan llamadas a la función *merge*. Cuando más gruesa sea la línea, más grandes son las dos secuencias que se funden.

3.4.3 Eficiencia

A partir de la observación del Algoritmo 3.9, se ve que la ordenación por fusión requiere un espacio auxiliar igual al tamaño de la entrada, $\Omega(n)$.

Por otra parte, en el código del algoritmo de ordenación por fusión no hay ninguna sentencia alternativa ni ningún bucle *while* que condicione el flujo. Esto permite asegurar que la eficiencia no tendrá distinción de casos.

Está claro que nos disponemos a utilizar el Teorema Maestro II para las recurrencias divisoras, ya que en las llamadas recursivas reducimos el tamaño del problema a la mitad. Hay dos llamadas que se ejecutarán siempre una tras otra. Eso es $a = 2$. Por otra parte, está bien claro que el tamaño del vector que se pasa por parámetro a *mergesort* se reduce a la mitad entre llamadas sucesivas, por tanto $b = 2$. Y entonces tenemos, como ya se ha dicho, que $T_{merge}(n) = \Theta(n)$. Eso quiere decir que $k = 1$. Luego entramos en el Teorema Maestro II por el caso en que $a = b^k$.

O sea, la eficiencia del algoritmo de ordenación por fusión es

$$T_{mergesort} = \Theta(n log(n))$$

en todos los casos. Es decir, independientemente de cómo estén ordenados inicialmente los valores a ordenar.

3.4.4 Variantes

De los algoritmos utilizados para ordenar, la ordenación por fusión es uno de los más antiguos. De ahí que haya habido muchas maneras de mejorarlo, hasta al punto que las primeras dos líneas del interior del bucle del Algoritmo 3.9 son conocidas como *copia de colas*. Un vistazo a alguna de estas mejoras. De entrada, en la función *merge* se podría remplazar la copia de colas por un par de centinelas, añadiendo los valores $a[n+1] = \infty$, y $b[m+1] = \infty$, en los finales de los vectores a y b. Eso ahorraría $2n$ comparaciones. Pegas. Disponer de los espacios $a[n+1]$ y $b[n+1]$ no siempre ocurre. Definir el valor ∞ podría no resultar fácil.

Otra enmienda que también se ha llevado a término es colocarse físicamente las dos secuencias de entrada de espaldas, de manera que la entrada a la función *merge* siempre sería un vector con un solo pico. Un contenido creciente al

inicio, y a partir del elemento l-ésimo, decreciente. Y también hay versiones que utilizan el mismo vector T como espacio auxiliar teniendo en cuenta la parte escrita y la disponible..., y complicaciones más sofisticadas. Ninguna de estas variantes impacta de una forma significativa en la eficiencia, y tan solo lo hacen en la constante oculta de la notación asintótica.

3.5 Algoritmos Históricos

Algunos algoritmos que pueden hacer gala de sintetizar ideas profundamente entramadas e innovadoras en unas pocas líneas de código se enmarcan en el esquema algorítmico de dividir y vencer. En esta sección se presentan dos de ellos, que hacen lucir la estrategia en su esplendor más espectacular. El primero, sirve para acelerar productos de números enteros cuando tienen muchas cifras. El segundo, para multiplicar matrices.

Cuando un alumno está en clase y aprende qué tipo de problemas pueden ser resueltos con la técnica de dividir y vencer, es fácil que decaiga por la astucia que hay detrás. Los algoritmos que se presentan a continuación no se descubren cada día. Y como siempre, lo que se dice es lo que se dice y no va más allá. En definitiva, como ya se ha apuntado anteriormente, en un control de la materia en el que entre este tema, normalmente nos van a pedir algoritmos que son variaciones de los de ordenación vistos anteriormente. Lo que sigue es puramente ilustrativo, y no se espera de nadie, que sea capaz de tener ideas como las que aquí se exponen un día cualquiera durante un par de horas.

3.5.1 Algoritmo de Karatsuba


Anatoly A. Karatsuba

Anatoly Karatsuba (1937-2008) fue un matemático ruso especializado en algoritmos eficientes entre otros ámbitos. Fue durante casi treinta años director del Departamento de Teoría de Números en el Steklov Institute of Mathematics. El año 1960 propuso un método para el problema de multiplicar números grandes, donde el tamaño de las instancias es el número de dígitos de los números que se trata de multiplicar. Este método de resolución de productos tuvo un impacto extraordinario en la historia del Algorítmia. Más concretamente en los esquemas algorítmicos. Se trata de uno de los paradigmas del esquema de dividir y vencer que nació conjuntamente con la definición del esquema algorítmico, de contenido más teórico. Y tanto o más importante es el algoritmo de la transformada rápida de Fourier, que no se muestra en este libro, pero es sin duda uno de los algoritmos que más influencia ha tenido en la historia de la computación, basado en el de Karatsuba que aquí es presenta.

El problema que se plantea es el cálculo del producto entre dos números. Sin pérdida de generalidad, supondremos que los dos números tienen la misma cantidad de cifras, ya que pueden tener ceros a la izquierda.

Dados dos números de n cifras, el algoritmo escolar consiste en multiplicar cada cifra del primero por cada cifra del segundo. Una vez calculadas todas las multiplicaciones entre las n^2 parejas posibles, hay $2n-1$ sumas adicionales. Luego, el tiempo necesario para realizar todas estas operaciones es $\Theta(n^2)$.

Para introducirse en el análisis que procede, vaya por delante un recordatorio de la formulación polinómica de un número. En términos formales, si tenemos un número d anotado como una secuencia de n dígitos, $d_{n-1}d_{n-2}\dots d_0$, entonces el valor numérico que expresa esta secuencia de dígitos se puede calcular como un polinomio de la base B en la cual el número esté expresado, con la equación

$$d = d_{n-1}B^{n-1} + d_{n-2}B^{n-2} + \cdots + d_0B^0,$$

o también

$$d = \sum_{i=0}^{n-1} d_iB^i.$$

Ya habíamos hablado de ello en la Sección 2.2 introduciendo los diccionarios. Esto es tan fácil de entender como que el número de 4 cifras 2753 es igual a $2 * 10^3 + 7 * 10^2 + 5 * 10^1 + 3 * 10^0$, siempre que sobreentendamos que lo estamos expresando en base $B = 10$.

Vista la formulación polinómica de los números en una base dada, incurrimos en un análisis breve del procedimiento de Karatsuba. Partimos de los dos números

$$x = x_{n-1}x_{n-2}\dots x_0$$
$$y = y_{n-1}y_{n-2}\dots y_0$$

siendo los x_i y los y_j, para $i, j \in \{0, \dots, n-1\}$, los dígitos de cada uno de ellos expresados en una base dada, B. Supongamos, para simplificar, que n es par. Entonces podemos decir a a la primera mitad del número x, la de más peso. De manera que $a = x_{n-1}x_{n-2}\dots x_{n/2}$. A la segunda mitad de x le llamamos b. O sea, $b = x_{(n/2)-1}x_{(n/2)-2}\dots x_0$. Lo mismo con el número y utilizando c y d, $c = y_{n-1}y_{n-2}\dots y_{n/2}$, y $d = y_{(n/2)-1}y_{(n/2)-2}\dots y_0$.

Llegados a este punto, ya podemos postular

$$xy = acB^n + (ad + bc)B^{n/2} + bd. \tag{3.1}$$

Y quedamos bastante satisfechos, porque ya tenemos una implementación utilizando el esquema algorítmico de dividir y vencer para resolver el problema de multiplicar dos números. Hemos conseguido transformar un procedimiento $\Theta(n^2)$, el algoritmo escolar, en este otro de la expresión (3.1), el tiempo del qual es $T(n) = 4T(n/2) + \Theta(n^k)$. Miremos si merece la pena.

Utilizamos una vez más el Teorema Maestro II. Tenemos el número de llamadas, $a = 4$, el factor de reducción, $b = 2$, y entonces, al no haber ningún otro bucle, $k = 0$. Caso $a > b^k$. Eficiencia $\Theta(n^{log_b(a)})$. Total, $\Theta(n^2)$. Cero patatero. No hemos ganado nada de nada.

Analizemos más minuciosamente el procedimiento escolar, haciendo una descripción formal de los actores. En particular, introducimos nuevos coeficientes, z_k, para $k \in \{0, \ldots, 2n-2\}$, que representan los coeficientes de la notación polinómica respeto la base B, para el valor resultante. O sea, el producto final será $z_{2n-2}z_{2n-1}\ldots z_0$.

En detalle,

$$
\begin{array}{cccccccc}
 & & & & & x_{n-1} & x_{n-2} & \ldots & x_0 \\
 & & & & & \times\ y_{n-1} & y_{n-2} & \ldots & y_0 \\
\hline
 & & & & & x_{n-1}y_0 & x_{n-2}y_0 & \ldots & x_0y_0 \\
 & & & & \ldots & & & & \\
 & & x_{n-1}y_{n-2} & x_{n-2}y_{n-2} & \ldots & & x_0y_{n-2} & & \\
+ & x_{n-1}y_{n-1} & x_{n-2}y_{n-1} & \ldots & & x_0y_{n-1} & & & \\
\hline
 & z_{2n-2} & z_{2n-3} & \ldots & & z_{n-1} & z_{n-2} & \ldots & z_0
\end{array}
$$

siendo $z_k = \sum_{i+j=k} x_i y_j$ para $k \in \{0, \ldots 2n-2\}$.

Este sumatorio no es trivial. Para cada k hay que coger todas las combinaciones de dos índices i y j tales que sumen k. O sea, para z_0, sólo x_0y_0, pero para z_1 ya es $x_0y_1 + y_1x_0$ y así sucesivamente.

En notación polinómica, y llamando z al valor del producto final. Podemos decir en una sola expresión

$$
\begin{aligned}
z = x_0y_0 + (x_0y_1 + y_1x_0)B + (x_0y_2 + x_1y_1 + x_2y_0)B^2 + \ldots \\
+(x_{n-1}y_{n-2} + x_{n-2}y_{n-1})B^{2n-3} + x_{n-1}y_{n-1}B^{2n-2} = \sum_{k=0}^{2n-2} z_k B^k.
\end{aligned}
$$

En definitiva, los coeficientes de la notación polinómica para el producto serán

$$
\begin{aligned}
z_0 &= x_0 y_0 \\
z_1 &= x_0 y_1 + x_1 y_0 \\
z_2 &= x_0 y_2 + x_1 y_1 + x_2 y_0 \\
&\dots \\
z_{2n-2} &= x_{n-1} y_{n-1}
\end{aligned}
$$

O sea, que volvemos a tener un nuevo problema de dividir y vencer planteado. Podemos resolver un producto de dos números grandes resolviendo $2n-2$ productos y unas cuantas sumas. No obstante, no sabríamos calcular la eficiencia de este problema, ya que el número de llamadas depende de n.

Bien, sigamos. Ahora viene lo interesante. Karatsuba aumenta la eficiencia porque resulta que hay cálculos de los z_k's que se pueden aprovechar.

Observémoslo con números de tan solo dos dígitos para aligerar la explicación. Tenemos así, solamente $z_0 = x_0 y_0$, $z_1 = x_0 y_1 + x_1 y_0$, y $z_2 = x_1 y_1$. Karatsuba propone expresar este z_1 del medio como

$$z_1 = x_0 y_1 + x_1 y_0 = (x_0 + x_1)(y_0 + y_1) + z_2 + z_0. \tag{3.2}$$

Y ahí la tenéis. Esta es la astucia de Karatsuba.

Por esta razón al principio de esta sección ya se decía que el alumno no ha de ponerse nervioso ante estas ideas excepcionales. A nadie, en un control, se le puede pedir tener ideas históricas.

Para poner un ejemplo fácil, multipliquemos 981 por 1234 que da 1210554. El tamaño del problema es $n = 4$. La parte de más peso sería claramente z_2 =09*12 = 108. La de menos, z_0 =81*34=2754. Y la parte central, con el algoritmo de la expresión (3.2) nos daría z_1 =90*46 + 108 + 2754 = 1278, ya que 09+81 = 90 (o sea $x_0 + x_1$ en (3.2)), y 12+34=46 (o sea $y_0 + y_1$). Total, el producto, 981 * 1234 = 108*10^4 + 1278*10^2 + 2754*10^0 = 1210554, fascinante.

Apuntamos la eficiencia del algoritmo de Karatsuba para la multiplicación de números. A partir de la expresión (3.2), tenemos que $a = 3$, ya que sólo hacemos 3 productos, que son z_0, z_2, y $(x_0 + x_1) * (y_0 + y_1)$. Entonces, $b = 2$ porque el número de dígitos de los números que multiplicamos es la mitad del tamaño inicial, n. Y $k = 1$, ya que el número de sumas necesarias aumenta linealmente con n. Y si $a > b^k$ como antes, ahora también como antes el tiempo resultante pertenece a $\Theta(n^{log_b(a)})$.

Sin embargo ahora, esto es $\Theta(n^{log_2\ 3})$, o lo que es lo mismo, $\Theta(n^{1.57\dots})$.

Aunque parezca poca cosa, hay que tener en cuenta que hay procesos que realizan productos de manera masiva, y si cada día tienes de calcular un millón, esta mejora puede ser muy importante.

La diferencia de calidades entre los algoritmos indicados en las expresiones (3.1) y (3.2) luce más cuanto más grande sea n. Por esta razón se llama algoritmo de multiplicación de números grandes de Karatsuba. Para entender el porqué, coged papel milimetrado y dibujaos las gráficas con las curvas de las dos funciones, n^2 y $n^{1.57}$, y ya vereis que tardan en separarse. Mejor que paper milimetrado quizás sería utilizar el gnuplot. A partir de 30 dígitos las diferencias son significativas.

En cualquier caso, y como ya se ha indicado más arriba, este algoritmo es un miembro constituyente del esquema algorítmico de dividir y vencer. Es muy curioso la manera de evolucionar que tiene el conocimiento teórico. Por un lado, el algoritmo de ordenación por fusión es anterior al de Karatsuba para la multiplicación. Por otro lado, el concepto de esquema algorítmico nació en tiempos del algoritmo de Karatsuba. Y es una vez formalizado el contenido teórico del esquema algorítmico cuando se incluyen ejemplos que existían antes que la teoría que ejemplifican. Dicho de otra manera, cuando John von Newmann hizo el mergesort estaba utilizando una técnica algorítmica pero él todavía no lo sabía, ya que aún no estaba formalizada. Después, surge un resultado espectacular como el de Karatsuba, y es en base a la espectacularidad de este resultado que se formaliza el esquema algorítmico de dividir y vencer. Bien, quizás no sólo de este. El ejemplo de la próxima sección también está en esta línea. Pero en cualquier caso, es interesante observar el proceso de formalización del conocimiento, similar a montarse una estantería cuando ya hay algunos libros desordenados por casa.

El Algoritmo de Karatsuba reduce el tiempo del producto entre dos números de n dígitos de $\Theta(n^2)$ a $\Theta(n^{1.57\cdots})$.

3.5.2 Algoritmo de Strassen


Volker Strassen

Volker Strassen (1936-...) es un matemático alemán ya retirado. Bien, matemático, físico, filósofo y músico. Habiéndose doctorado en temas de estadística, su carrera hizo un giro hacia el análisis de algoritmos.

El año 1969 propuso un método para multiplicar matrices de manera más eficiente que $\Theta(n^3)$. De hecho, llevó las cosas más allá hasta conseguir un algoritmo para la inversión rápida de matrices no esparsas. De todas formas, aquí se muestra el primero, que también ha sido el más citado en la literatura.

La disquisición que sigue habla de matrices. Se refiere a matrices cuadradas, y excepcionalmente, en lugar de llamar n al tamaño de la instancia que sería el número de elementos de las matrices, denotaremos por n el número de filas, o de columnas, ya que a lo largo de la sección completa se habla tan solo de matrices cuadradas. Utilizar n para significar el número de filas en las matrices es más antiguo que utilizarla para medir las instancias de los problemas. Por tanto, hacemos un paréntesis en nuestra notación por respeto a los antecesores.

Comenzamos el análisis echando un vistazo a un algoritmo muy conocido. Es bien sabido que el algoritmo clásico de multiplicación de matrices, para matrices cuadradas, puede ser descrito en términos simbólicos como se muestra a continuación.

$$\begin{pmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,n} \\ & \dots & & \\ a_{n,1} & a_{n,2} & \dots & a_{n,n} \end{pmatrix} \begin{pmatrix} b_{1,1} & b_{1,2} & \dots & b_{1,n} \\ b_{2,1} & b_{2,2} & \dots & b_{2,n} \\ & \dots & & \\ b_{n,1} & b_{n,2} & \dots & b_{n,n} \end{pmatrix} = \begin{pmatrix} c_{1,1} & c_{1,2} & \dots & c_{1,n} \\ c_{2,1} & c_{2,2} & \dots & c_{2,n} \\ & \dots & & \\ c_{n,1} & c_{n,2} & \dots & c_{n,n} \end{pmatrix}$$

donde

$$c_{1,1} = a_{1,1}b_{1,1} + a_{1,2}b_{2,1} + \cdots + a_{1,n}b_{n,1}.$$

Y, más en general, $c_{ij} = \sum_{k=1}^{n} a_{ik}b_{kj}$, $\forall i, j \in \{1, \dots, n\}$.

En el Algoritmo 3.12 tenemos una implementación de este procedimiento. El código computa el producto $C = AB$, siendo tanto A como B matrices de $n \times n$ valores reales.

```
void producto_de_matrices(int n, double* C[], double* A[], double* B[])
{
    for (int i=0; i<n; i++) {
        for (int j=0; j<n; j++) {
            C[i][j] = 0.0;
            for (int k=0; k<n; k++) C[i][j] = C[i][j] + A[i][k] * B[k][j];
        }
    }
}
```

Algoritmo 3.12 *Algoritmo clásico de multiplicación de matrices cuadradas.*

No es difícil ver que el tiempo de esta operación pertenece a $\Theta(n^3)$.

Introduzcámonos, como en la sección anterior en un análisis más minucioso de las operaciones. Consideraremos, sin pérdida de generalidad, que n es una potencia de 2. Eso es $n = 2^k$ para alguna k entera. Si las matrices no tienen estas dimensiones, rellenamos con ceros las posiciones de abajo o de la derecha necesarias.

Comencemos. Se trata de multiplicar dos matrices A y B para obtener otra, C de las mismas dimensiones.

$$C = AB \qquad A, B, C \in \mathbb{R}^{n \times n}$$

Entonces, segmentamos A, B, y C en cuatro bloques del mismo tamaño todos ellos.

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} , B = \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix} , C = \begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix}$$

donde $A_{i,j}, B_{i,j}, C_{i,j} \in \mathbb{R}^{n/2 \times n/2}$. Podemos calcular el producto de las matrices grandes A y B multiplicando estos bloques como si fueran simples números, aunque sean matrices. Tenemos que

$$\begin{aligned} C_{1,1} &= A_{1,1}B_{1,1} + A_{1,2}B_{2,1} \\ C_{1,2} &= A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\ C_{2,1} &= A_{2,1}B_{1,1} + A_{2,2}B_{2,1} \\ C_{2,2} &= A_{2,1}B_{1,2} + A_{2,2}B_{2,2} \end{aligned}$$

Hasta aquí, ya hemos conseguido un algoritmo para la multiplicación de matrices que utiliza el esquema de dividir y vencer. Igualmente, pero, necesitamos $8 = 2^3$ productos de tamaño mitad del problema inicial. Esto sigue siendo $\Theta(n^3)$.

Pero de nuevo, nos encontramos con una genialidad. Resulta que si definimos las siguientes 7 matrices,

$$\begin{aligned} M_1 &= (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2}) \\ M_2 &= (A_{2,1} + A_{2,2})B_{1,1} \\ M_3 &= A_{1,1}(B_{1,2} - B_{2,2}) \\ M_4 &= A_{2,2}(B_{2,1} - B_{1,1}) \\ M_5 &= (A_{1,1} + A_{1,2})B_{2,2} \\ M_6 &= (A_{2,1} - A_{1,1})(B_{1,1} + B_{1,2}) \\ M_7 &= (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2}) \end{aligned}$$

entonces podemos conseguir el producto de las dos matrices iniciales con las reglas siguientes

$$
\begin{aligned}
C_{1,1} &= M_1 + M_4 - M_5 + M_7 \\
C_{1,2} &= M_3 + M_5 \\
C_{2,1} &= M_2 + M_4 \\
C_{2,2} &= M_1 - M_2 + M_3 + M_6.
\end{aligned}
$$

Este algoritmo también es miembro constituyente del esquema algorítmico.

El algoritmo de Strassen para el producto de matrices cuadradas reduce el tiempo de cómputo de $\Theta(n^3)$ a $\Theta(n^{2.58\cdots})$.

Con el capítulo de dividir y vencer se han cubierto diferentes aspectos de la teoría del Algorítmia. Por un lado se ha formalizado el concepto de esquema algorítmico, uno de los pilares del algorítmia computacional. También se ha hecho un análisis en profundidad de los algoritmos de ordenación que no habían sido estudiados antes de este capítulo. Y con estos, hemos completado el estudio de los algoritmos de ordenación más conocidos. Hemos dejado otros en el tintero, pero es que cuando se habla de teorías tan abiertas (una ordenación se puede hacer de muchas maneras), nunca se puede hacer un repaso exhaustivo de todos los enfoques que puede tener el problema.

Hemos visto que ordenar n elementos es un proceso que tarda $\Theta(n\ log(n))$ en todos los algoritmos de ordenación serios.

Este $\Theta(n\ log(n))$ no nos debe sorprender. Era previsible, ya que podemos interpretar una ordenación como n búsquedas de las posiciones finales donde colocar cada valor, y cada búsqueda tarda $\Theta(log(n))$. Como se ve, la notación asintótica es una herramienta que en muy pocos símbolos sintetiza un proceso más o menos sofisticado. Qué y cómo, o cuántos y cuales.

Conviene recordar que el mejor algoritmo de ordenación, en el caso peor, es el peor algoritmo de ordenación. La eficiencia del quicksort es de $\Theta(n^2)$ cuando el vector de entrada ya está ordenado.

Finalmente se ha presentado un par de algoritmos que en su momento crearon escuela. Son ideas extraordinarias que mejoran procesos clásicos. Esto ha servido para dar identidad al esquema algorítmico de dividir y vencer.

Capítulo 4

Grafos

En este capítulo abriremos nuevos horizontes. No es como el capítulo anterior, que procurábamos realizar tareas tan eficientemente como fuera posible. No. Ahora se trata de aprovisionarnos de artillería para las nuevas guerras que vendrán. En capítulos posteriores nos enfrentaremos a problemas que no sabremos resolver. Es necesaria una buena reserva de conocimiento para procurar hacer lo que podamos.

Los grafos sirven para modelar situaciones (que se dice pronto), expresar precedencias, mostrar relaciones, considerar decisiones, ... y no acabaríamos de encontrar utilidades. Un grafo es, ciertamente, una estructura mental de una utilidad que va más allá de cualquier síntesis. Es difícil explicar para qué sirve un grafo. Es casi imposible incluir todas sus aplicaciones en alguna estructura verbal. Sin duda, sí que hay formas de explicar el concepto de grafo, como que es una herramienta para expresar relaciones entre conceptos. Pero no nos movemos de ahí mismo. Un nivel de abstracción muy alto. Un nivel de abstracción casi tan alto como el de conjunto. La única manera que tenemos de comprender todo aquéllo para lo que puede ser útil el concepto de grafo es con la experiencia. O ninguna, quizás no hay manera y nunca en la vida lleguemos a comprender todas los usos que puede tener un grafo.

El comportamiento de un autómata se modela con un grafo. Un organigrama es un grafo. Un diseño de una base de datos o un diagrama de flujo. Un mapa de carreteras, una carretera, un circuito electrónico, un trozo de cable, una jugada futbolística, un desplazamiento, cualquier red de comunicación, de distribución, o de producción. Problemas de exploración, de localización, de enrutamiento, o de flujos. Todo aquéllo que tiene cierta complejidad, aquéllo que cuesta pensar... y los problemas que no sabemos, o no se pueden, resolver.

El capítulo comienza con la definición formal de grafo. Se recuerda Leonhard Euler y se enuncia algo de léxico y algún teorema. Después, veremos las

estructuras de datos informáticas que utilizaremos para representarlos.

Cuando tengamos definido el grafo como estructura de datos, veremos qué procedimientos necesitamos para llevar a cabo las operaciones que parecen más elementales: Los recorridos. Así, en plural. Dos recorridos son diferentes si varía la secuencia ordenada de los vértices que visitan. Visitar un vértice significa tratarlo, hacer cualquier cosa con él.

Acabaremos el capítulo introduciendo una cuantificación. Asociaremos pesos a las aristas. Esto nos abrirá la utilidad de esta estructura a un amplísimo abanico de problemas que también veremos en capítulos posteriores.

4.1 Definición

En la teoría de conjuntos, se establece que un conjunto no puede contener elementos repetidos. Para definir un conjunto que pueda tener elementos repetidos, hay que utilizar el concepto de multiconjunto.

Definición 4.1 Grafo. *Decimos que $G = G(V, E)$ es un grafo si V es un conjunto de elementos, y E un multiconjunto de parejas de elementos de V.*

Llamamos *vértices* o *nodos* a los elementos del conjunto V. Son tan protagonistas del tema, que les dedicamos dos palabras sinónimas.

Podría parecer que un grafo tiene una definición con una cierta redundancia. Parece que únicamente diciendo las parejas, ya quedase dicho el conjunto de vértices, y por tanto, que no sea necesario poner ese conjunto en la definición. O sea, si un grafo está formado por parejas de elementos, y cada pareja viene identificada por los dos identificadores de sus vértices, entonces solamente nombrando estas parejas parece que tenga que bastar.

Pero indiscutiblemente, lo que constituye un grafo y no depende de ninguna otra cosa son los vértices, V. Un vértice representa un punto, un concepto, un individuo,... alguna cosa referenciable. Las parejas, en cambio, dependen de los vértices. Los dos vértices de cada elemento del multiconjunto E tienen que ser necesariamente del grafo.

Parece lógico pensar, pues, que cuando identificamos un grafo $G(V, E)$, el conjunto V sólo aparezca por los vértices *aislados* que puedan haber, aquéllos no incidentes a ninguna arista. Y no parece tan lógico pero es más cierto, que el conjunto V aparece porque, intuitivamente, es lo bastante fundamental como para aparecer en la definición.

Cuando E no tiene elementos repetidos, el grafo se llama *simple*. Es frecuente hablar de grafos suponiendo que se habla de grafos simples. Entonces se dice que E es un conjunto. Y aún así, todo lo que se dice, muchas veces también vale cuando E es un multiconjunto tal como dice la definición. Esto es así por comodidad del lenguaje. Resulta más cómodo utilizar el término *conjunto* que *multiconjunto*.

Bien, a las parejas o elementos del multiconjunto E los llamamos de distinta forma si son parejas ordenadas o no.

4.1.1 Grafos No Dirigidos

Cuando los elementos del conjunto E no tienen un orden implícito, se denominan *aristas* (*edges* en inglés). Al grafo $G = G(V, E)$ se le califica con el adjetivo negado de *no dirigido*. Siendo $u, v \in V$ dos vértices aparejados por una arista, denotamos por $e = \{u, v\} \in E$ la arista que los enlaza.

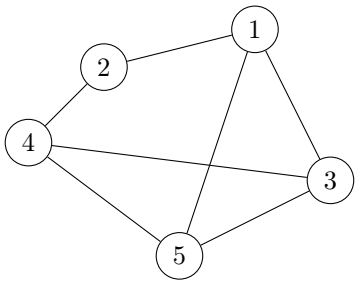


Figura 4.1: *Grafo no dirigido.*

Para el grafo no dirigido de la Figura 4.1 tenemos,

$$\begin{aligned} V &= \{1, 2, 3, 4, 5\}, \\ E &= \{\{1, 2\}, \{1, 3\}, \{1, 5\}, \{2, 4\}, \{3, 4\}, \{3, 5\}, \{4, 5\}\}. \end{aligned}$$

Siempre que no hay confusión posible, llamamos a la arista e simplemente como $e = uv$. Es decir, podemos referirnos a una arista por sus nodos, $uv \in E$, siendo $u, v \in V$.

Usamos el verbo *incidir* de una manera commutativa. Se dice que una arista incide en cada uno de sus dos nodos. También se dice que un nodo incide en las aristas que lo contienen. También es habitual decir que una arista *relaciona*, *conecta*, o *une* sus dos vértices.

Decimos que los *vecinos* de un vértice $v \in V$, son aquéllos $u \in V$ tales que existe la arista $e = uv \in E$. Al conjunto de vecinos de un nodo se le llama *adyacencia* del nodo. Es lo mismo decir adyacencia de v que vecinos de v.

$$adj(v) = \{u \in V | uv \in E\}, \qquad \forall v \in V.$$

El término *corte* de un nodo está ínitimamente relacionado con el conjunto de vecinos. El corte de un nodo v es el conjunto de aristas que inciden en él. Se le acostumbra a denotar $\delta(v)$. Gráficamente, es el conjunto de aristas que deberíamos cortar si quisiéramos sacar el vértice del grafo.

$$\delta(v) = \{e \in E | e = vu, u \in V\}, \qquad \forall v \in V.$$

Al número de elementos del corte de un nodo, $|\delta(v)|$, que está bien claro que coincide con el número de vecinos, $|adj(v)|$, se le llama *grado* del nodo. Y la cosa va más allá. Si un nodo tiene un grado par, entonces se le llama directamente nodo *par*. Y si el grado es impar, nodo *impar*. Esta terminología es intensamente utilizada en problemas de enrutamiento por grafos.

4.1.2 Grafos Dirigidos

Cuando los elementos del conjunto E tienen un orden concreto de los dos posibles, entonces los llamamos *arcos*. En ese caso se califica de *dirigido* al grafo $G = G(V, E)$. También se le denomina *dígrafo*, y así mismo se puede expresar como $D = D(N, A)$, siendo N el conjunto de vértices o nodos. Denotamos los arcos con paréntesis, indicando que efectivamente hay un orden entre los dos vértices, $e = (u, v) \in A$.

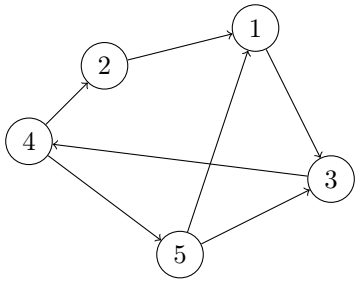


Figura 4.2: *Grafo dirigido o* dígrafo.

Para el grafo dirigido de la Figura 4.2 tenemos,

$$\begin{aligned} N &= \{1, 2, 3, 4, 5\}, \\ A &= \{(1,3), (2,1), (3,4), (4,2), (4,5), (5,1), (5,3)\}. \end{aligned}$$

Decimos que un arco $a = uv \in A$ *va de* un nodo $u \in N$ *a* otro $v \in N$. También decimos que el nodo u es la *cola* u *origen* del arco uv, y v es la *cabeza* o *destino*. Un arco, pues, va de su cola a su cabeza. La adyacencia de un vértice $v \in N$ en un dígrafo es la unión entre los conjuntos de *predecesores* y *sucesores* del vértice.

$$\begin{aligned} pred(v) &= \{u \in N | (u, v) \in A\}, \qquad \forall v \in N, \\ succ(v) &= \{u \in N | (v, u) \in A\}, \qquad \forall v \in N, \\ adj(v) &= pred(v) \cup succ(v). \end{aligned}$$

En estos grafos se puede distinguir entre corte *de entrada* de un nodo, $\delta^-(v)$ y corte *de salida* $\delta^+(v)$. También se define el corte $\delta(v) = \delta^-(v) \cup \delta^+(v)$, y se habla de grado *entrante* y grado *saliente*, siendo el grado del nodo la suma de estos dos grados.

4.1.3 Orden, Tamaño, y Otra Terminología

El vocabulario propio de la teoría de grafos es profuso. Como se ha mencionado, se dice que un grafo es simple si no hay aristas repetidas. Se dice que un grafo es *completo* si es simple y tiene una arista entre cada pareja de nodos. Si un grafo es completo, sólo diciéndonos el número de vértices ya lo tenemos todo, ya que si es dirigido tendrá $n(n-1)$ arcos, y si no, $n(n-1)/2$ aristas. El grafo completo de n nodos se acostumbra a denotar por $\mathcal{K}_n$.

En la teoría de grafos se denomina *orden* al número de vértices $n = |V|$, lo cual es una lástima ya que en el análisis de algoritmos nos gusta llamarle n al tamaño de las instancias. Lo que se llama *tamaño* en la teoría de grafos es el número de aristas, $m = |E|$. Si asumimos que el grafo es simple, entonces $0 \leq m \leq n(n-1)/2$ para los grafos no dirigidos, y $0 \leq m \leq n(n-1)$ si son dirigidos. O sea, los dos términos, *orden* y *tamaño*, son muy adecuados y muy ajustados a lo que expresan. Está claro que *tamaño* tiene una connotación más precisa que *orden*, que recuerda más a aproximación. En rigor, el término *orden* indica una magnitud asociada a un máximo (como ocurre con la notación asintótica cuando decimos $O(n)$, por ejemplo). Todo eso cuadra con que el tamaño, definido como el número de aristas, da más precisión de lo grande que es el grafo que el orden, definido como el número de vértices, siempre que no haya una gran cantidad de vértices aislados. Definimos los parámetros métricos de un grafo como si nunca sucediera que haya muchos vértices aislados. Está bien. De hecho, no ocurre casi nunca. Y cuando efectivamente nos encontramos con grafos que tienen altos porcentajes de vértices aislados, acostumbran a ser grafos resultantes de operaciones secundarias en procesos intermedios, y no forman parte del grueso que domina la eficiencia de los algoritmos.

En definitiva, nos cuadra que se le llame *tamaño* al número de aristas, y *orden* al número de vértices. Lo que no nos gusta tanto es que se le llame m al tamaño.

Cuando más similares sean m y $n(n-1)/2$ en un grafo no dirigido, o m y $n(n-1)$ en uno de dirigido, más *denso* será el grafo. En otras palabras, cuantas más aristas o arcos tenga un grafo, más denso será. Denso es lo contrario de *esparso*.

Sea $G = G(V, E)$ un grafo no dirigido. Decimos *camino*, de longitud k, entre dos nodos v_0 y v_k pertenecientes a V, a una secuencia $v_0 e_1 v_1 e_2 v_2 \dots e_k v_k$ tal que cada $e_i = \{v_{i-1}, v_i\} \in E$, $i = 1, \dots, k$, y no haya nodos repetidos, $v_i \neq v_j$, $\forall i, j \in \{0, \dots, k\}$. Como no hay vértices repetidos, tampoco puede haber aristas repetidas. Si en un camino el primer nodo es el mismo que el último, $v_0 = v_k$, entonces lo llamamos *ciclo*. Luego, un ciclo no es un camino, ya que tiene algún nodo repetido. Si un grafo no tiene ningún ciclo, entonces es un grafo *acíclico*.

Un grafo que para cualquier pareja de vértices existe un camino de uno otro, es un grafo *conexo*. Cuando esto no ocurre, el grafo tiene varias *componentes conexas*, es decir, subconjuntos de vértices entre los que existen los caminos de unos a los otros. Un grafo no dirigido, acíclico y conexo es un *árbol*. Observad que los árboles son los únicos grafos conexos con menos aristas que nodos. Un árbol siempre tiene $n-1$ aristas. A diferencia de árboles estudiados en capítulos anteriores, estos árboles no tienen un nodo especial que sea la raíz. A menudo, se considera cualquier nodo como tal. En la Figura 4.3 se puede ver un grafo conexo, un árbol, y un grafo disconexo con dos componentes conexas.

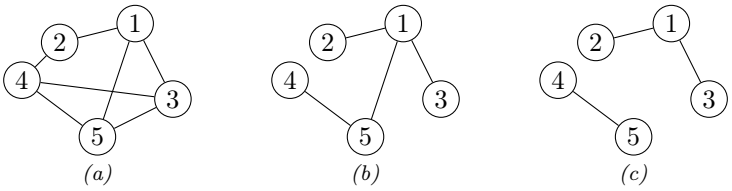


Figura 4.3: *(a) Grafo conexo; (b) Árbol; (c) Grafo disconexo con dos componentes.*

Para el caso de los grafos dirigidos la cosa se complica. La definición de camino es fácilmente extensible, siempre y cuando tengamos en cuneta que cada arco del camino ha de empezar en la cola y acabar en la cabeza. Respeto a la conectividad de los grafos dirigidos, aparecen por fin algunos adverbios. Un grafo dirigido puede ser *fuertemente* conexo, si desde cualquier vértice es posible ir a cualquier otro vértice. Si no, puede ser *unilateralmente* conexo, si para cualquier pareja de vértices se puede ir o bien de un al otro o bien del otro al uno. Y si no, todavía puede ser *débilmente* conexo, que significa que si le quitásemos las direcciones a los arcos, el grafo no dirigido equivalente quedaría conexo.

En la Figura 4.4 se puede ver la diferencia entre los tres tipos de conectividad definidos para los grafos dirigidos. El de la Figura 4.4*(c)* no es ni siquiera unilateralmente conexo, ya que no se puede ir ni del 3 al 2 ni viceversa.

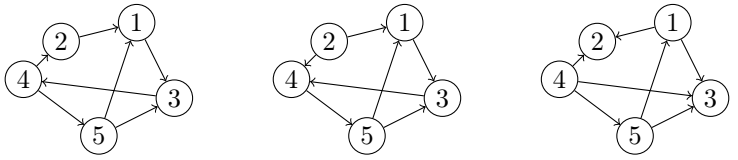


Figura 4.4: *(a) Fuertemente conexo; (b) Unilateralmente conexo; (c) Débilmente conexo.*

4.1.4 Teoremas

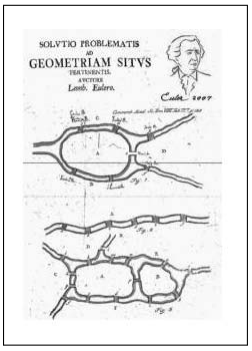
En esta sección recordamos a un Gran Mestro, vemos algunas propiedades básicas de grafos y echamos un vistazo a los fundamentos de la teoría que los alberga. De todo ello, nos quedaremos con algún postulado que más tarde utilizaremos en los cálculos de eficiencia.

Leonhard Euler

Leonhard Euler (1707-1783) fue un matemático y físico nacido a Basilea, Suiza. Incuestionablemente uno de los matemáticos más reconocidos de la historia, fundó lo que ahora se conoce como la teoría de grafos. También definió el concepto de función tal como la conocemos hoy día, pilar fundamental de las disciplinas de cálculo y análisis matemático. A lo largo de su vida, se fue quedando ciego, cosa que no parece demasiado trascendente si tenemos en cuenta su producción intelectual. Se dice que era capaz de escribir de memoria epopeyas latinas de Virgilio, y de recordar las seis primeras potencias de los primeros cien números primos. Un fuera de serie que merece el más grande de los respetos cuando no veneración. A Euler, sin duda, le debemos gran parte de lo que sabemos y de nuestra manera de entender el mundo. El fue el introductor del número e como base de la función exponencial. Y otras maravillas..., nos dejó una de las fórmulas trigonométricas más bonitas que recordamos,

$$e^{2\pi i} + 1 = 0,$$

o, dicho de otra manera, $e^{i\pi} = \sqrt{-1}$, siendo $i^2 = -1$ la esencia de los números imaginarios constituyentes de los números complejos. Esta fórmula es fruto de que para cualquier ángulo φ, $e^{i\varphi} = cos(\varphi) + i\ sin(\varphi)$, como sale impreso en el sello.



Se cuenta que Leonhard Euler fue consultado sobre el dilema de los siete puentes de Königsberg. Y también que este fue el hecho a partir del cual nace la teoría de grafos. Desde 1735, los habitantes de Königsberg saben que no pueden salir de casa dando un paseo que cruce por sus siete puentes, sin repetir ninguno, si quieren volver a casa. Y también ahora ya sabemos todos, Euler nos lo explicó, que el número de vértices impares en cualquier grafo del mundo es par. Repitámoslo. El número de vértices impares en cualquier grafo es par. O sea cero, dos, cuatro,... etcétera.

Y también hemos aprendido que

- Si hay cero vértices impares, o sea, todos los vértices pares, un grafo se puede recorrer sin pasar dos veces por la misma arista acabando en el vértice inicial.
- Si el número de vértices impares es dos, entonces se puede recorrer todo sin pasar dos veces por la misma arista, siempre que se empiece en un vértice impar y se termine en el otro.
- Y si un grafo tiene cuatro, seis, ocho, o más vértices impares, entonces es imposible pasar por todas las aristas sin repetir ninguna.

Por otra parte, también es normal asociar Euler con poliedros. No hay que forzar la imaginación para entender que los poliedros son representaciones de grafos. De ahí la terminología de vértices y aristas.

Con todo, que trascienda en lo que aquí nos ocupa, está el postulado de que la suma de los grados de todos los nodos es el doble del número de aristas. Eso nos interesa a la hora de calcular eficiencias. Se puede intuir una demostración por inducción sobre el número de aristas de este teorema.

4.2 Representación

En esta sección definiremos estructuras de datos para representar grafos. Visto está que los vértices de un grafo forman un conjunto, y que un conjunto no puede tener elementos repetidos. O sea, un grafo no puede tener vértices repetidos. Es decir, los vértices han de ser identificables. Que en un conjunto los elementos sean identificables significa que existe una operación lógica de comparación (=) que dados dos elementos del conjunto nos retorna un valor lógico, que semánticamente nos indica si son iguales o no. Esta operación debe ser reflexiva, simétrica, y transitiva. Hasta aquí no hay nada de nuevo. Ahora pero, requerimos una nueva condición. Los vértices han de ser ordenables.

Si además de identificables son ordenables, entonces existe una operación lógica adicional ($<$) que dados dos elementos responde si el primero es menor que el segundo. Esta operación no ha de ser reflexiva ni simétrica, pero sí transitiva.

Total, si en un grafo no tuviéramos un orden definido sobre los vértices, lo deberíamos inventar para poderlo representar.

En esta sección se trata de hacer disponible una estructura de datos a la que le podamos pedir acciones propias de un grafo. Por simplicidad, se muestran las implementaciones tan peladas como es posible. Suponemos que los grafos que almacenaremos en estas estructuras tienen los vértices numerados del 1 al n.

La estructura *grafo* contiene dos operaciones constructivas, *anadir_vertice()* y *anadir_arista(u,v)*. Como ejemplo de otras operaciones adicionales, definiremos también la función booleana *hay_arista(u,v)*. Una definición más completa de esta estructura de datos debería incluir operaciones destructivas como *borrar_vertice* o *borrar_arista*. Aquí se muestra el código justo para poder comprobar los procedimientos que se verán en secciones posteriores. También definiremos dos macros que usaremos en cabeceras de bucles. Una para poder recorrer todos los vértices de un grafo, *para_todo_vertice(u,g)*, y la otra para todos los vecinos de un nodo, *para_todo_vecino(v,g[u])*.

Seguidamente se presentan las dos implementaciones más populares de los grafos.

Primero la más rígida, la *matriz de adyacencias*, que es más ágil para las operaciones básicas. De hecho, la implementación que se presenta aquí para las matrices de adyacencias es una implementación totalmente estática que sólo admite grafos de hasta cien nodos, muy sencilla. Esta implementación es más útil para grafos muy densos. Cuanto más denso sea el grafo, más adecuada es su implementación en matrices de adyacencias. Por esta razón se limita el número de nodos del grafo, o sea, el orden.

Después la más flexible, *listas de adyacencia*, aunque para las operaciones básicas no sea tan eficiente como la implementación con la matriz, tiene una capacidad indefinida. No hay límite al número de nodos. No obstante, si el grafo es denso las operaciones se ralentizan notablemente. Por todo esto, es la implementación indicada para grafos esparsos.

Desde un punto de vista más filosófico, hay una analogía entre la dualidad de la matriz de adyacencias versus las listas de adyacencia, y la dualidad entre los problemas directos versus problemas inversos, ya mencionados en la Sección 3.4.1, cuando se ha visto la ordenación por fusión. Un ejemplo de esta misma dualidad reside en el corazón de la informática gráfica entre las tecnologías ráster y vectorial. ¿Qué diferencia hay entre un dibujo y una foto?.

Tal vez convenga aclarar que una imagen ráster es aquélla que asigna un color

a cada punto del plano rectangular que rellena. Estos puntos se llaman píxeles, contracción de *picture element*. O sea, la cantidad de espacio que requiere la imagen no depende de los objetos que estén presentes en ella.

Una imagen vectorial, en cambio, es una secuencia de elementos geométricos (redonda, cuadrado, línea, punto,...) descritos en coordenadas del plano que rellenan, de manera que es necesario un visualizador que sea capaz de interpretar estos elementos y dibujarlos, para poderla ver. Existe un lenguaje para esa descripción. Y cuantos más objetos aparezcan en el dibujo, más espacio de memoria ocupará la imagen. La enorme ventaja que tiene la tecnología vectorial es que la calidad de la imagen no se deteriora con los cambios de escala, zooms y demás.

Es más fácil rasterizar una imagen vectorial que vectorizar un imagen ráster. Imagináoslo.

Y ahora el debate, ¿Cuál es más eficiente? la tecnología vectorial o la tecnología ráster?.

Hombre, si nos referimos a una imagen donde aparezcan muchas cosas, entonces la foto, o sea la tecnología ráster. Vendría a ser como con un grafo muy denso, casi completo, que resultaría más eficiente implementado en una matriz de adyacencias. Si, al contrario, hablamos de una imagen donde haya tan sólo un círculo, entonces el dibujo es más eficiente. Este vendría a ser como un grafo esparso implementado en listas de adyacencia.

Respeto a esa dualidad entre las tecnologías de la informática gráfica, el mundo ráster tiene una ventaja importante. Las pantallas de los ordenadores utilizan la tecnología ráster. Aún así, puede resultar interesante saber que veinte años atrás existían pantallas vectoriales, monocromáticas de fósforo verde, por supuesto. Jamás volveremos a ver las diagonales tan finas y precisas que esos monitores proyectaban. Eran líneas con una precisión tal que no se podían borrar, y entonces, la única manera para poder dibujar en la pantalla de nuevo era desconectándolas de la alimentación eléctrica, desenchufarlas, y esperar un ratito por la persistencia que quedava. En cualquier caso, cuando hacían líneas, nunca se producía el efecto escalita (*aliaising*, como le llaman algunos) tan desagradable, al que ahora todos nos hemos tenido que acostumar cuando pintamos rayas inclinadas en una pantalla.

Al tema, la dualidad entre las dos implementaciones nace de considerar a priori si el grafo que representarán será muy denso o muy esparso.

Si suponemos que será muy denso, entonces nos guardamos una información para cada posible arista. De esta manera, cada arista real se alojará en un espacio indexable y todo irá más rápido. Pero claro, si después resulta que el grafo no es denso, estaremos ocupando mucho espacio para las aristas posibles que no existen, y eso hará la implementación ineficiente espacialmente. Estaríamos matando moscas a cañonazos. Si suponemos de entrada que el grafo tendrá

unas poquitas de las aristas posibles, entonces lo más eficiente será apuntarnos en alguna lista las aristas que existen, igual que se hace con una base de datos. Procediendo por esta vía, si el grafo resulta ser muy denso, las operaciones serán muy lentas.

4.2.1 Matriz de Adyacencias

Representamos un grafo en una matriz cuadrada M de dimensiones $n \times n$, los valores de la cuál son $m_{u,v} \in \{0, 1\}$ si el grafo es simple. De forma que $m_{u,v}$ indicará si hay una arista entre los vértices u y v. En términos formales viene a ser

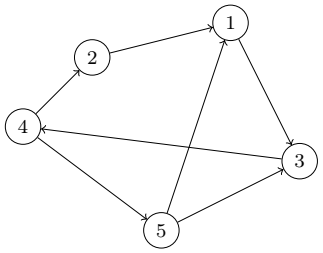
$$M : [1, n] \times [1, n] \to \{0, 1\}.$$

para el caso de grafos simples. De lo contrario podemos extender los números $m_{u,v}$ al conjunto de los naturales, representando el número de veces que la arista uv aparece en el grafo, su multiplicidad. Tendríamos,

$$M : [1, n] \times [1, n] \to \mathbb{N}.$$

Si el grafo es no dirigido, la matriz M será simétrica, $m_{u,v} = m_{v,u}$, $\forall u, v \in \{1, \ldots, n\}$.

En la Figura 4.5*(a)* se muestra un grafo dirigido y en la Figura 4.5*(b)* una representación gráfica de su implementación en una matriz de adyacencias.



(a)

		1		
1				
			1	
	1			1
1		1		

(b)

Figura 4.5: *(a) Grafo dirigido; (b) Representación en matriz de adyacencias.*

Como se ha mencionado, la implementación mostrada en el Algoritmo 4.1 es casi un implementación de juguete. No se ha utilizado una estructura semiestática para evitar la gestión de memoria con punteros a punteros. Si pudiésemos saltar adelante, aquí convendría utilizar las estructuras de vectores dinámicos y matrices dinámicas que se explican en la Sección 6.2. En este punto todavía se puede evitar complicar los algoritmos, y en fin, tal como se presenta la clase resulta lo bastante útil para poder seguir las explicaciones que se dan a lo largo de este capítulo.

```
#include <memory.h>
#define N 100

class grafo_matriz {
      int n;
      int M[N][N];
      bool dirigido;

      void crea() {n=0; memset(matriz,0,N*N*sizeof(int)); dirigido = false; }
      void crea(bool d) { crea(); dirigido = d; }

public:
      grafo_matriz(bool d) { crea(d); }

      void anadir_vertice() {
          n++;
      }

      void anadir_arista(int u, int v) {
          matriz[u][v] = 1;
          if (!dirigido) matriz[v][u] = 1;
      }

      bool hay_arista(int u, int v) { return matriz[u][v] == 1; }
};
```

Algoritmo 4.1 *Declaración de la clase para la implementación de un grafo en una matriz de adyacencias.*

En el Algoritmo 4.1, el constructor establece si el grafo es dirigido o no, e inicializa la matriz a ceros con la instrucción *memset*. Al ser una estructura estática no requiere destructor. Las otras operaciones son del todo triviales.

El análisis de la eficiencia de esta implementación es similar al de la Sección 2.2.1 cuando se veía la implementación de los diccionarios en un vector. Todas las operaciones son $\Theta(1)$. Eso es fantástico. Sin embargo, también igual que en aquella sección, la limitación de espacio provoca esta estructura resulte impracticable para grafos medianamente grandes. Sólo de arrancar, un programa que utilice la clase *grafo_matriz* mostrada en el Algoritmo 4.1 ya ocuparía 40 Kb. El problema, pues, es que guardamos un espacio para cada arista posible, tanto si existe como si no. Este puñado de ceros ocupan demasiado espacio, tan sólo para darnos la información de una negación (que no existe la arista en cuestión), y eso es muy ambicioso. No se puede pretender saber cada cosa que no ocurre.

La eficiencia espacial de las matrices de adyacencia es $\Omega(n^2)$, cosa que a

menudo las convierte en inadmisibles.

4.2.2 Listas de Adyacencia

Representamos un grafo en un vector de listas. Cada posición del vector representa un vértice, y la lista asociada a esa posición, la lista de sus vecinos. Ésta es una implementación dinámica. Su flexibilidad nos permite no tener que limitar el número de vértices como en el caso anterior.

En la Figura 4.6*(a)* se muestra un grafo no dirigido y en la Figura 4.6*(b)* una representación gráfica de su implementación en listas de adyacencia.

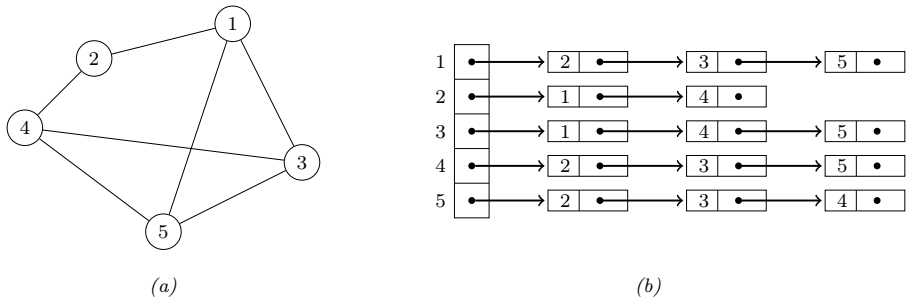


Figura 4.6: *(a) Grafo no dirigido; (b) Representación en listas de adyacencia.*

Antes que la definición de la clase para el grafo debemos definir las listas que contendrá el vector. Así pues, éste es un buen momento para repasar la estructura *lista*. En el Algoritmo 4.2 tenemos una nueva implementación de la estructura nodo que formará las listas. Cada cajita de la Figura 4.6*(b)* vendrá implementada en una estructura como ésta.

```
struct nodo {
    int v;
    nodo* siguiente;
    nodo(int u) { v = u; siguiente = NULL; }
};
```

Algoritmo 4.2 *Estructura de datos* nodo *para a la formación de las listas.*

Un nodo es una estructura que tan solo contiene un entero, que será el vértice que representa, y una referencia a un posible nodo siguiente. La única manera

de asignar un valor a un nodo es en el momento de la creación. Y no se puede crear un nodo si no se le da algún valor.

La estructura *lista* implementada en el Algoritmo 4.3 no debería mostrar ninguna cosa nueva para el nivel de conocimientos de programación que se le supone al lector. Como se ve, utiliza la estructura *nodo* del Algoritmo 4.2. La referencia de un nodo igual a NULL es final de lista.

```
struct lista {
    int n;
    nodo* primero;

    void crea() { n = 0; primero = NULL; }
    void destruye(nodo* p) {
        if (p && p→siguiente) destruye(p→siguiente); }
        delete p;
    }
    void destruye() { destruye(primero); }
    void pon(int u) {
        nodo* nuevo = new nodo(u);
        nuevo→siguiente = primero;
        primero = nuevo; n++;
    }
    void quita() {
        nodo* antiguo = primero;
        primero = primero→siguiente;
        delete antiguo; n--;
    }
};
```

Algoritmo 4.3 *Estructura de datos* lista.

Es una estructura bien sencilla con las operaciones indispensables. Únicamente tiene un contador de elementos, un apuntador, y las operaciones para poner y quitar un elemento. Para que resulte tan pelada como sea posible, notad que los elementos se ponen siempre en la primera posición de la lista, y también se quitan siempre de la primera posición. Esta lista es una pila. Así, tenemos que todas las operaciones son $\Theta(1)$. En el código que acompaña el libro, se puede comprobar que las inserciones de aristas en el grafo están en orden decreciente con el fin de que las listas queden ordenadas ascendientemente.

Bien, pasemos ya a la clase que implementa un grafo en listas de adyacencia. La clase *grafo_listas* implementada en el Algoritmo 4.4 consta de tres variables miembro privadas. El número de vértices n, el vector de listas *vector*, y un valor lógico por si se trata de un grafo dirigido o no.

El constructor y el destructor son públicos, lógicamente, aunque las tareas que realizan son privadas. Al constructor le decimos si el grafo es dirigido o no, cosa que servirá para añadir aristas. El destructor invoca la destrucción de cada lista, la función recursiva mostrada en el Algoritmo 4.3, y finalmente libera el espacio ocupado por el vector de apuntadores.

```
class grafo_listas {
      int n;
      lista* vector;
      bool dirigido;
      void crea() { vector = NULL; n=0; dirigido=false; }
      void crea(bool d) { crea(); dirigido = d; }
      void destruye() {
          for (int i=1; i<=n; i++) vector[i].destruye();
          delete [] vector;
      }

public:
      grafo_listas(bool d) { crea(d); }
      ~grafo_listas() { destruye(); };

      void anadir_vertice() {
          lista* aux = new lista[1+n+1];
          for (int i=1; i<=n; i++) aux[i] = vector[i];
          delete [] vector;
          n++;
          aux[n].crea();
          vector = aux;
      }

      void anadir_arista(int u, int v) {              // 1≤ u,v≤n
          vector[u].pon(v);                           // no dirigido→ u>v
          if (!dirigido && u>v) anadir_arista(v,u);
      }

      lista& operator[](int i) { return vector[i]; }
      int tamano() { return n;}

      bool hay_arista(int u, int v) {
          para_todo_vecino(l,vector[u]) if (l→v == v) return true;
          return false;
      }
};
```

Algoritmo 4.4 *Declaración de la clase para la implementación de un grafo en listas de adyacencia.*

Se sigue con las dos funciones que caracterizan la estructura como grafo. Por sencillez, hay errores probables que no se controlan. Estos errores se comentan en las mismas descripciones de las dos funciones que vienen a continuación.

Por una parte *anadir_ vertice()*. Esta función comienza reservando el espacio necesario para guardar $1+n+1$ listas. El primer 1 es porque no queremos usar el primer índice del vector. Indexamos a partir del índice 1 para que coincida con el nombre del primer vértice. Quizás en el futuro cambiaremos de perspectiva y llamaremos cero al primer vértice del grafo. De momento, suena mal. En el espacio de las n listas siguientes guardaremos las listas de adyacencia, y el segundo 1 es el correspondiente al nuevo vértice que estamos añadiendo. Una vez disponible el espacio, la rutina copia los n valores de los apuntadores actuales en el nuevo vector. Entonces libera el espacio ocupado por el vector actual que ya es antiguo. Incrementa el número de elementos, y establece el final de lista en el nuevo vértice. Con todo, es $\Theta(n)$. Podría hacerse de formas más eficientes, pero por el nivel del código que se muestra, esta implementación parece lo bastante pragmática.

Y después tenemos la función *anadir_ arista(u,v)*. Se comenta en el código las dos precondiciones. Por un lado un código más robusto debería controlar que los dos números unidos por una arista formasen parte del conjunto de números posibles. Es decir, se asume que u y v son mayores o iguales a 1 y menores o iguales a n. Además, también es precondición del procedimiento que si el grafo es no dirigido, u sea mayor que v, cosa que tampoco se controla. Con todo, nos queda una rutina bien sencilla que tan solo crea un nuevo nodo con el valor del segundo vértice de la arista, y la añade como primer elemento de la lista del primero. Esta función es $\Theta(1)$.

También se ha añadido un operador embellecedor. Solamente sirve para aumentar la legibilidad del código. Así se podrá hacer referencia a la lista asociada a un vértice concreto mediante el uso de claudátors [].

Sigue la función *tamano()* que es un envoltorio público para la variable miembro n. Esta función se usa, por ejemplo, en la definición de las macros para hacer los recorridos del grafo que se muestran en el Algoritmo 4.5.

```
#define para_todo_vertice(a,b) for (int a=1; a<=b.tamano(); a++)
#define para_todo_vecino(a,b) \
            for (nodo* a=b.primero; a!=NULL; a = a→siguiente)
```

Algoritmo 4.5 *Definiciones para hacer recorridos en la esctructura grafo_ lista.*

Finalmente hay la función booleana *hay_ arista(u,v)*, que hace uso de la segunda macro definida. Esta función no es tan eficiente como en el caso de las matrices de adyacencias. Es $\Theta(m)$, siendo $m = |E|$.

Tal como se ha explicado en la Sección 4.1.3, para hablar de eficiencia en el caso de los grafos nos volvemos a encontrar con el inconveniente que nos habíamos topado en el algoritmo de Strassen de la Sección 3.5.2. Históricamente, la n en la teoría de grafos se ha utilizado para el número de nodos, $n = |V|$. Hasta ahí vale. El hecho es que el número de nodos de un grafo no es el tamaño del grafo, sinó el orden. Es lógico. Un grafo completo, si es dirigido, tiene $n(n-1)$ arcos. Eso nos indica que una vez conocido el número de nodos de un grafo simple tenemos como máximo un número de aristas determinado.

Evitando alimentar la confusión pues, utilizaremos como notación los valores $|V|$ y $|E|$. Pero para no hacer la lectura tan retorcida pondremos directamente los conjuntos. Así, $O(V)$ representa directamente $O(|V|)$.

Y en general, cuando el argumento de la notación asintótica sea un conjunto, entenderemos que nos estamos refiriendo al cardinal del conjunto. Con lo dicho, la eficiencia espacial de un grafo implementado en listas es $\Omega(V + E)$. Eso significa que, siempre que el grafo tenga más aristas que vértices, el espacio que necesita esta representación es $\Theta(m)$, o sea del lineal con el tamaño del grafo, cosa que ya nos gusta. Lo que pasa es que este tamaño no se llama n, sinó m.

4.3 Recorridos y Exploraciones

Las estructuras de datos vistas en la Sección 4.2 sirven tan solo para grafos *explícitos*.

El concepto de grafo trasciende su representación. En la resolución de ciertos problemas, las representaciones que se han visto en la Sección 4.2 no nos valen. Ni estas representaciones, ni ninguna otra. Hay problemas que para solucionarse requieren grafos tan grandes que no los podemos almacenar en memoria. O sea, hay dos tipos de grafos. Los que caben en memoria y los podemos representar, que llamamos grafos explícitos, y los que no caben, de los que tan solo podemos representar una parte, que denominamos grafos *implícitos*.

Para los grafos explícitos pues, tenemos un recorrido sencillísimo aunque no respeta la topología del grafo pero que para muchas aplicaciones es suficiente. Simplemente se trata de recorrer la estructura con las dos macros del Algoritmo 4.5. Con un recorrido así de sencillo podemos resolver problemas sencillos. Por ejemplo, queremos saber si un grafo no dirigido implementado en listas tiene algún vértice aislado. Entonces tan solo recorriendo el vector y mirando si hay alguna lista vacía ya podemos responderlo.

Ahora bien, a menudo interesa recorrer el grafo moviéndonos tan solo por los nodos y las aristas que lo constituyen. Para estos casos tenemos dos recorridos ortogonales. El recorrido en anchura, y el recorrido en profundidad. Estas dos operaciones sirven tanto para grafos explícitos como para grafos implícitos

immensamente grandes. Cuando iniciamos un recorrido en un grafo implícito a partir de algún nodo conocido, entonces utilizamos la palabra *exploración* en lugar de recorrido.

Las exploraciones utilizan un subconjunto de aristas del grafo que exploran. Este conjunto de aristas forma un árbol, el *árbol de exploración*. Ya se ha visto en capítulos anteriores que la implementación de un árbol se puede hacer en un vector de predecesores. Para eso conviene definir cualquier nodo como raíz. Hagamos memoria, un vector de predecesores es un vector en el cual cada índice representa un nodo y cada contenido el nodo padre. Por ejemplo, en la Figura 4.7 se puede ver un árbol y su vector de predecesores. Para este vector, se ha decidido enraizar el árbol en el nodo 5.

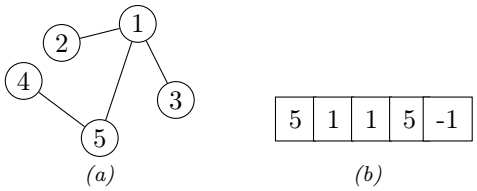


Figura 4.7: *(a) Árbol; (b) Vector de predecesores con nodo raíz* 5.

Los procedimientos que realizan las exploraciones o recorridos reciben como parámetros de entrada el grafo y un vértice inicial $s \in V$ (s de *start*). Este nodo inicial se convierte en la raíz del árbol de exploración, que puede ser uno de los parámetros de salida del procedimento.

Para poder incidir en diversos instantes de los procedimientos, en las exploraciones establecemos dos fases por lo que respecta a la visita de los nodos. Distingiremos entre *abrir* un nodo, y *cerrar*lo. Entonces, a lo largo de una exploración los nodos podrán estar en tres estados diferentes.

- estado *desconocido*: El algoritmo todavía no ha tenido conocimiento de la existencia del nodo. Pintamos el nodo blanco.
- estado *abierto*: El algoritmo ya ha considerado el nodo, pero todavía hay vecinos de los que no se tiene conocimiento. Vecinos en estado desconocido. Pintamos el nodo gris.
- estado *cerrado*: El algoritmo ha visitado este vértice completamente. Todos sus vecinos estan abiertos o cerrados, pero no tiene ningún vecino en estado desconocido. Pintamos el nodo negro.

Según qué tipo de operación deseemos hacer en el recorrido, nos interesará actuar en el momento de abrir un nodo, o en el de cerrarlo. En el código que acompaña este libro, las funciones *abro(v)* y *cierro(v)* de los Algoritmos 4.9 y 4.11 están implementadas en el archivo del programa principal de este capítulo. Son funciones para poder tratar los nodos en el momento que nos interese. La

única cosa que hacen las del código librado es imprimir un mensaje por pantalla diciendo si se está abriendo o cerrando, y el nombre del nodo. Así se puede observar la secuencia de tratamientos para cada recorrido.

Los dos recorridos que se ven seguidamente recorren el grafo topológicamente. Eso significa que a partir de un vértice tienen conocimiento de sus vecinos y nada más. Por esta razón, sólo son capaces de explorar una componente conexa. Si queremos hacer una exploración de un grafo disconexo, de varias componentes, entonces necesitaremos usar una función como la que se muestra en el Algoritmo 4.6.

```
void recorrer(grafo_listas& g, int modo)
{
    int n = g.tamano();
    C = new color[1+n];
    memset(C,blanco,(n+1)*sizeof(color));
    para_todo_vertice(u,g) {
        if (C[u] == blanco) {
            if (modo == BFS) bfs(g,u);
            if (modo == DFS) dfs(g,u);
        }
    }
    delete [] C;
}
```

Algoritmo 4.6 *Módulo externo auxiliar para recorrer grafos disconexos.*

Esta función recibe como parámetros el grafo a explorar, y un indicador de si se quiere utilizar el recorrido en anchura (BFS), o el recorrido en profundidad (DFS). Hace uso de un vector dinámico C, declarado en el espacio global de la librería, para guardar el estado de los vértices. Este puntero está declarado en un espacio global. Si en lugar de presentar estas funciones bajo la técnica de programación imperativa pronfundizásemos en las clases definidas en la sección anterior, entonces el apuntador C, en lugar de estar en el espacio global de la librería podría ser una variable miembro adicional de la clase. Además, nos quitaríamos de encima el parámetro g ya que el código estaría dentro de la clase. Pero en este libro se quiere presentar algoritmos de las formas más diversas.

Inicialmente, la rutina del Algoritmo 4.6 reserva el espacio para el vector de colores, y pone todos los nodos como desconocidos, en el *memset*. Y entonces entra en un bucle governado por una condición. Precisamente, el número de veces que esta condición sea cierta coincide con el número de componentes conexas del grafo. Bien, se comienza, pues, con todos los vértices blancos, y por tanto se entra en la condición con el primer vértice del grafo. Entonces se llama a *bfs* o *dfs* que retorna con tantos vértices negros como haya en la componente conexa

del primer nodo, ya que este es el primero que se ha utilizado de raíz. Si queda algún nodo blanco, entonces hay otras componentes conexas.

Hay un juego muy divertido para ordenar las cartas de un juego de naipes. Se empieza poniendo encima de la mesa los naipes boca abajo, tapados. Todas las cartas bien puestas en disposición reticular de 4 filas $\times$ 12 columnas. Con naipes españoles. Oros, copas, espadas y bastos. Una vez las cartas están todas tapadas y en posición matricial, cogemos una cualquiera, por ejemplo la primera de arriba a la izquierda, posición (1,1), donde debería ir el as de oros. La giramos y nos aparece el siete de espadas, por ejemplo. Entonces cogemos la carta de la posición (3,7). La giramos para mirar cuál es y en su lugar dejamos el siete de espadas boca arriba, que es la que corresponde a la posición. Y aparece el dos de copas. Con el dos de copas en la mano cogemos la carta de la posición (2,2) y la giramos, y en el lugar que ocupa dejamos el dos de copas. Así podemos ir siguiendo el hilo hasta que al girar alguna carta, nos encontramos el as de oros por fin. Esto provoca que nos quedemos sin carta para girar. Para seguir el juego debemos coger otro naipe de alguna posición inventada que todavía esté boca abajo, claro. En este juego se gana cuando menos veces tengas que inventarte qué carta girar. De hecho, quizás no es tan divertido. Este juego es un solitario para niños pequeños y muy aburridos, ya que, tal vez no lo hayáis observado pero el resultado no depende en absoluto de ninguna estrategia que pueda hacer el jugador. Casi es tan aburrido como tirar un dado al aire y que se gane cuanto más alto sea el valor que salga.

Bien, en cualquier caso, cada vez que tenemos la mala suerte de que salga la carta que llena el espacio vacío, se corresponde con un retorno de la rutina *bfs(g,u)* o *dfs(g,u)* del Algoritmo 4.6. Una carta tapada que por casualidad estuviera colocada en su lugar de la matriz sería como un nodo aislado en el recorrido en profundidad. Una nueva componente conexa, en definitiva.

```
void recorrer(grafo_listas& g, int modo, int bosque[])
{
    int n = g.tamano();
    C = new color[1+n];
    memset(C,blanco,(n+1)*sizeof(color));
    para_todo_vertice(u,g) {
        if (C[u] == blanco) {
            if (modo == BFS) bfs(g,u,bosque);
            if (modo == DFS) dfs(g,u,bosque);
        }
    }
    delete [] C;
}
```

Algoritmo 4.7 *Módulo externo auxiliar para recorrer grafos disconexos obteniendo los árboles de exploración.*

Para poder tener información de todas las componentes conexas del grafo que se explora, deberíamos transformar el Algoritmo 4.6 tal como se muestra en el Algoritmo 4.7. Este parámetro *bosque*, cuando efectivamente haya más de una componente conexa, resultará un vector con varias raíces, varios valores a -1, que representarán los árboles de exploración de cada componente. Observad que se deja la responsabilidad de la gestión de la memoria para este nuevo vector al módulo que llama a la exploración.

4.3.1 Recorrido en Anchura *BFS*

El recorrido en anchura (en inglés, *breadth first search*) es un recorrido local. Eso significa que explora los nodos por orden de distancia a la raíz. Primero los vecinos. Después, los vecinos de estos vecinos. Y después los vecinos de estos últimos. Para plasmar la idea en un trazo, conviene asociar el recorrido en anchura a una espiral, Figura 4.8.

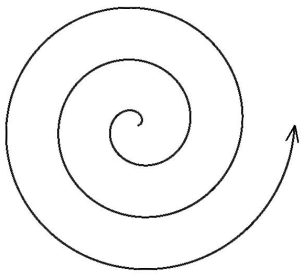


Figura 4.8: *Imagen mnemotécnica de la exploración en anchura.*

Con esta idea se pretende sintetizar el concepto de exploración local. La invariante característica de esta exploración es que se visitan todos los nodos a distancia k antes que cualquier otro a distancia $k+1$, de la raíz. Por esta razón, hablando de exploración en anchura nos viene a la cabeza la idea de barrer. La exploración en anchura realiza un barrido del grafo.

La propuesta que se muestra en esta sección es una propuesta iterativa. Ya se puede intuir, pues, que para esta exploración se utilizará una cola donde se añadan los vecinos desconocidos del nodo actual. Este es un buen momento para echar una ojeada a una estructura de datos bien sencilla que hará las funciones de cola para al recorrido que seguidamente se verá.

En el Algoritmo 4.8 tenemos el código que implementa esta estructura. Se trata tan solo de un apuntador a un entero y un número de elementos. El constructor tan solo inicializa las dos variables miembro. Como está implementada en un vector dinámico es necesario un destructor que libere el espacio usado.

```
struct cola {
    int* Q;
    int n;

    cola() { Q = NULL; n = 0; }
    ~cola() { delete [] Q; }

    void anadir(int v) {
        int* nuevos;
        nuevos = new int[n+1];
        memcpy(nuevos,Q,n*sizeof(int));
        delete [] Q;
        nuevos[n] = v;
        Q = nuevos;
        n++;
    }

    int primero() {
        int v = Q[0];
        int* nuevos = new int[n-1];
        memcpy(nuevos,&Q[1],(n-1)*sizeof(int));
        delete [] Q;
        Q = nuevos;
        n--;
        return v;
    }

    bool vacia() { return n==0; }
};
```

Algoritmo 4.8 *Estructura de datos* cola.

Finalmente, las dos funciones que caracterizan una cola. Por un lado *anadir(v)* que añade un número entero que representará un vértice, al final del vector. Para hacer esto, en primera instancia reserva la memoria necesaria, es decir, para guardar los n elementos que hay actualmente en la cola más el elemento nuevo que se está añadiendo. Después usa una instrucción *memcpy* que copia los n elementos que tenía la cola antes de esta adición en el nuevo espacio, que se supone disponible. Entonces se libera el espacio ocupado por el apuntador antiguo, se añade el nuevo elemento en el espacio ahora disponible, y se actualiza el valor del apuntador a la nueva memoria acabada de adquirir y de llenar. La función *primero()* de la estructura *cola* del Algoritmo 4.8 extrae el primer elemento de la cola y lo retorna. Se lo guarda en una variable local para poderlo retornar. Después actúa exactamente a la inversa que la función *anadir()*. Como se puede ver en el Algoritmo 4.8, también se ha implementado una función booleana indicando si la cola está vacía, es decir, si el número de elementos es igual a cero.

Provistos con la estructura para la cola, ya podemos desvestir el procedimiento de la exploración en anchura. En el Algoritmo 4.9 se presenta un recorrido en anchura de un grafo. Como ya se ha dicho, los vértices pueden estar en tres estados que los simbolizamos con blanco (desconocido), gris (abierto), y negro (cerrado).

Comenzamos la exploración marcando todos los vértices del grafo como desconocidos menos el nodo inicial, s. Al nodo s que será la raíz del árbol de exploración resultante, lo marcamos como abierto. Una vez abierto, lo añadimos a la cola y entramos en un bucle que acabará cuando la cola esté vacía. Se cumple una invariante: Siempre, los vértices que formen la cola son vértices grises, vértices abiertos.

```
void bfs(grafo_listas& g, int s)
{
    int n = g.tamano();
    C = new color[1+n];
    memset(C,blanco,(n+1)*sizeof(color));
    C[s] = gris; abro(s);

    cola Q;
    Q.anadir(s);
    while (!Q.vacia()) {
        int u = Q.primero();
        para_todo_vecino(l,g[u]) {
            int v = l→v;
            if (C[v] == blanco) {
                C[v] = gris; abro(v);
                Q.anadir(v);
            }
            C[u] = negro; cierro(u);
        }
    }
    delete [] C;
}
```

Algoritmo 4.9 *Exploración en anchura (BFS).*

Una vez dentro del bucle tomamos el primer vértice de la cola, que en la primera iteración será el mismo nodo s, y abrimos todos los vecinos que hasta ahora sean desconocidos. En la primera iteración lo serán todos. De hecho, cuando nos encontremos que algún vecino no es blanco, o sea, cuando no se cumpla la alternativa, entonces es que hemos encontrado un ciclo. Mira qué bien, las exploraciones de los grafos sirven para encontrar ciclos. Cuando finalmente ya hemos abierto todos los vecinos de un nodo, entonces lo cerramos. Lo ponemos en negro.

Con el procedimiento del Algoritmo 4.9 recorreríamos una componente conexa del grafo, cosa que es de pura lógica. Si estamos haciendo una exploración del grafo en base a su topología, entonces no hay forma posible de saltar de una componente a otra. Tal como hemos resuelto esta cuestión en el Algoritmo 4.7, recurrimos en última instancia a la estructura que tengamos, la matriz o el vector de listas. O sea, de alguna manera estamos asumiendo que los grafos implícitos, los que son muy grandes, son conexos. Si no fuera así, entonces deberíamos conocer como mínimo un nodo de cada componente que queramos explorar.

Hay una repetición entre los Algoritmos 4.7 y 4.9. La reserva de memoria para el vector C. Eso es debido a que los algoritmos que se muestran en este libro pretenden ser independientes. Respeto a eso, deberíamos aclararnos. Si sabemos que sólo queremos recorrer una componente conexa, entonces utilizamos el Algoritmo 4.9 tal y como está. En cambio, si no sabemos el número de componentes conexas de un grafo que queremos recorrer completamente, entonces utilizando el Algoritmo 4.7, deberíamos sacar las líneas tercera y la penúltima del Algoritmo 4.9 .

De las exploraciones, se acostumbra a extraer el bosque, o árbol. Normalmente en un vector de predecesores. También es frecuente definir la *distancia* entre dos vértices, que significa el camino más corto que los une. Dos vértices vecinos están a distancia 1. Igual que los árboles de exploración, los vectores de distancias son informaciones que se pueden extraer de un recorrido.

En el Algoritmo 4.10, para conseguir el vector de distancias se haría un tratamiento idéntico al del árbol, añadiéndolo a la cabecera, que quedaría

void bfs(grafo_listas& g, int s, int p[], int d[]).

Igualmente sería necesario inicializarlo a ceros, añadiendo una instrucción como

*memset(d,0,(n+1)*sizeof(int));*

detrás del *memset* que hay actualmente. Y además, también deberíamos hacer el cálculo propiamente dicho de las distancias de cada nodo. Eso sería añadir la instrucción

d[v] = d[u] + 1;

bajo la asignación $p[v] = u$;.

La responsabilidad de la reserva y la liberación de memoria por lo que respecta a informaciones adicionales se deja en manos del módulo que invoque el recorrido.

```
void bfs(grafo_listas& g, int s, int p[])
{
    int n = g.tamano();
    C = new color[1+n];
    memset(C,blanco,(n+1)*sizeof(color));
    C[s] = gris; abro(s);

    memset(p,-1,(n+1)*sizeof(int));

    cola Q;
    Q.anadir(s);
    while (!Q.vacia()) {
        int u = Q.primero();
        para_todo_vecino(l,g[u]) {
            int v = l→v;
            if (C[v] == blanco) {
                C[v] = gris; abro(v);
                Q.anadir(v);
                p[v] = u;
            }
        }
        C[u] = negro; cierro(u);
    }
    delete [] C;
}
```

Algoritmo 4.10 *Exploración en anchura con obtención de información adicional.*

Aplicando el procedimiento del Algoritmo 4.10 al grafo no dirigido de la Figura 4.1 obtendríamos un árbol como el de la Figura 4.9*(a)*.

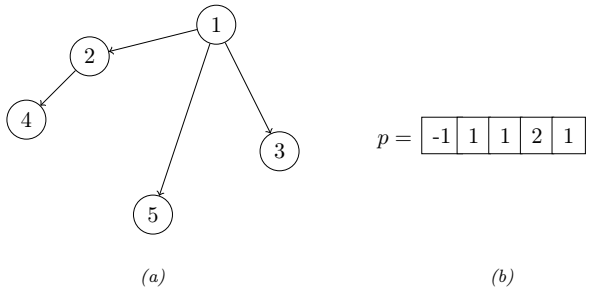


Figura 4.9: *(a) Árbol de exploración $T_{bfs}(G)$ del grafo de la Figura 4.1.; (b) Valores del vector que implementa el árbol.*

La implementación en un vector de predecesores del árbol de la Figura 4.9*(a)* se muestra en la Figura 4.9*(b)*.

Imaginaos que estáis en un castillo desconocido, en el hall justo habiendo cruzado la puerta de entrada. Hay un tesoro. Probablemente os pondríais a buscarlo explorando el castillo. Estaríais en un lugar donde habría unas cuantas puertas y alguna escalera. Comenzaríais estableciendo un orden entre todos los accesos. Por ejemplo primero las puertas de la planta baja de izquierda a derecha. Entonces cruzaríais la primera puerta y pasaríais a otra cámara que podría tener unas cuantas puertas más. Pues bien, haciendo un recorrido en anchura, antes de cruzar cualquiera de estas nuevas puertas, primero volveríais al recibidor y abriríais la siguiente puerta según el orden establecido. De manera que las nuevas estancias que hubieren más allá de la cámara vista después de cruzar la primera puerta, no serían exploradas hasta después de haber estado en cada una de las habitaciones accesibles cruzando sólo una puerta desde el recibidor. En el fondo, para un castillo desconocido, no es la mejor manera de hacerlo... El recorrido en anchura tiene una discontinuidad topológica cada vez que acabamos una lista de vecinos.

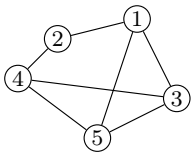
En la Figura 4.10 se puede ver la sucesión de estados de los nodos para el mismo grafo del ejemplo, a lo largo de toda la exploración del Algoritmo 4.10. No obstante, la relación entre las figuras y las líneas de código no queda explícita. Sí que se puede asociar la Figura 4.10*(1)* al estado justo después de la quinta línea, después del *memset*. También es correcto decir que la Figura 4.10*(2)* es justo después de la línea siguiente. Y a partir de ahí, las siguientes figuras son ya iteraciones del bucle principal.

Paso a paso. Respiremos hondo. En la Figura 4.10*(3)* hemos entrado dentro del bucle principal, hemos extraído el primero de la cola, hemos entrado a explorar su adyacencia, y hemos descubierto su primer vecino, el 2. En la Figura 4.10*(4)*, el segundo, el 3. Y en la 4.10*(5)* hemos descubierto el último vecino del 1.

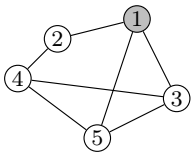
En la misma figura, parte *(6)*, el vértice 1 ya ha sido cerrado de color negro. Entonces, que en la figura no se manifesta, hay que recordar que el primer vecino del 1 que se ha añadido a la cola ha sido el 2. En la Figura 4.10*(7)*, que ya nos encontramos a la siguiente iteración, el nodo 2 es el nodo que obtenemos de la cola en la operación *primero()*. Entramos a abrir su adyacencia, y tan solo el vértice 4 queda por descubrir. Eso provoca que sea gris y que se inserte a la cola. Por otra parte, el vértice 2 ya se cierra, porque no queda ningún otro vecino por descubrir. Entonces recordamos que en la cola, después del 2 se ha añadido el 3. Por tanto, en la Figura 4.10*(8)*, ya en la iteración siguiente, no ocurre nada de nuevo. Todos los vecinos del vértice actual ya han sido abiertos. El nodo 3 se cierra sin que ni una sola vez haya sido cierta la sentencia alternativa. En la Figura 4.10*(9)* ya aparece cerrado. Una vez cerrado el nodo 3 hay que recordar, o si lo preferís, mirar la *(5)*, para ver que en la cola, después del nodo 3 se ha insertado el 5. Por tanto, en la Figura 4.10*(10)* se cierra el nodo 5 antes que el nodo 4, que ha sido el último en ser añadido a la cola, y por tanto se cierra finalmente, en la Figura 4.10*(11)*. Hecho.

El orden en el cuál los nodos se cierran, a partir de la Figura 4.10*(9)*, es importante, aunque no quede ningún vértice por descubrir. Los recorridos pueden

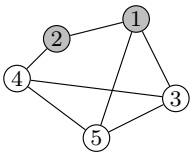
servir para múltiples objetivos. Y está claro que la finalidad en muchos casos no tiene por qué ser la simple visita de cada uno de los nodos, sinó otros objetivos dependientes del problema. Si en lugar de un esquema iterativo se hubiera descrito un esquema recursivo, entonces se debería distinguir entre recursividad de ida y de vuelta. En otras palabras, los momentos de apertura y cierre de los vértices, y el orden en que se efectúan estas operaciones, tiene un impacto directo en las aplicaciones que utilizan estos recorridos.



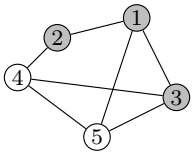
(1) Todos desconocidos.



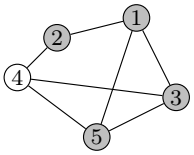
(2) Abrimos nodo 1.



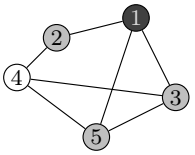
(3) Abrimos vecino → 2.



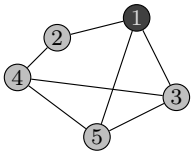
(4) Abrimos vecino → 3.



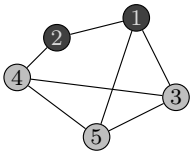
(5) Abrimos vecino → 5.



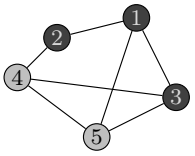
(6) Cerramos nodo 1; Nodo actual 2.



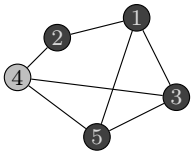
(7) Abrimos vecino → 4.



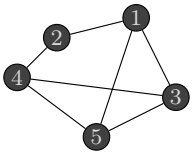
(8) Cerramos nodo 2.



(9) Cerramos nodo 3.



(10) Cerramos nodo 5.



(11) Cerramos nodo 4.

Figura 4.10: *Evolución de marcas en los nodos en un recorrido en anchura.*

En un breve análisis de eficiencia, se puede asegurar que el bucle principal no se ejecutará más de n veces, ya que la condición booleana que gobierna el flujo será cierta una vez para cada vértice. Eso es $\Theta(V)$. Por otra parte, como se ha dicho en la Sección 4.1.4, la suma de los grados de todos los vértices es $\Theta(E)$. Es importante que quede claro eso de la suma de grados de todos los nodos. Contemos cuántas veces se ejecutará, en total, el bucle *para_ todo_ vecino()* del Algoritmo 4.10. La primera vez, será el número de vecinos de la raíz, que no sabemos cuántos son. La segunda, el número de vecinos del primer vecino de la raíz, que tampoco sabemos cuántos serán. Y así toda el rato. No sabemos cuáles seran los números que debemos ir sumando, pero sabemos que todos ellos sumados darán un resultado igual a $2|E|$, que es $\Theta(E)$. Así pues, resulta lo que

era de esperar.

La eficiencia de un recorrido en anchura es $\Theta(V + E)$.

4.3.2 Recorrido en Profundidad *DFS*

El recorrido en profundidad (en inglés, *depth first search*) es un recorrido ortogonal al recorrido en anchura. Eso significa que explora el grafo profundizando tanto como sea posible en cada camino que abre. El orden de apertura y cierre de los nodos adopta una disposición apilada. Primero se abre el nodo raíz de la exploración, después el primer vecino en el orden prestablecido. Después, el primer vecino de este vecino. Y después el primer vecino del primer vecino del primer vecino. Así hasta encontrar un nodo terminal, que no tenga ningún otro vecino que el que nos ha llevado a él, su padre en el árbol de exploración. Entonces, este nodo se cerrará, y se abrirá el siguiente vecino de su padre.

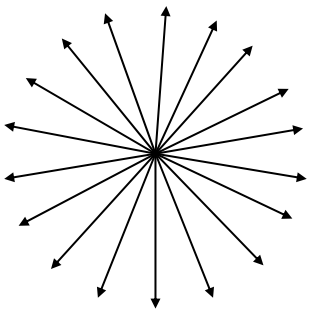


Figura 4.11: *Imagen mnemotécnica de la exploración en profundidad.*

Para plasmar la idea en un símbolo, conviene asociar el recorrido en profundidad a una estrella, Figura 4.11. Con esa idea se pretende sintetizar el concepto de exploración en profundidad. La invariante característica de esta exploración es que se visitan todos los nodos accesibles a través del vecino k antes que cualquiera accesible a través del vecino $k+1$. El recorrido en profundidad es un recorrido intrépido, valiente, que va tan lejos como es posible de la manera más rápida posible. Por eso, hablando de exploración en profundidad nos viene a la cabeza el concepto de sonda. Explorar en profundidad es sondear el grafo.

El algoritmo de exploración en profundidad rastrea el grafo con la máxima continuidad topológica posible. Cada nodo que se abre no se cierra hasta que se hayan abierto, y después cerrado, todos los descendientes en el subárbol de exploración con raíz en el nodo en cuestión.

La esencia de la exploración en profundidad se muestra en el Algoritmo 4.11. Es una implementación recursiva. Eso nos ahorra utilizar estructuras de datos auxiliares como la cola de la exploración en anchura. La misma pila del sistema

que nos soporta la anidación de llamadas nos sirve para controlar el orden en que se visitan los vértices.

```
void dfs(grafo_listas& g, int s)
{
    C[s] = gris; abro(s);
    para_todo_vecino(l,g[s]) {
        int v = l→v;
        if (C[v] == blanco) dfs(g,v);
    }
    C[s] = negro; cierro(s);
}
```

Algoritmo 4.11 *Exploración en profundidad (DFS).*

Se supone que las funciones de los Algoritmos 4.11 y 4.12 se llaman desde la función implementada en el Algoritmo 4.7. O sea, que aquí no se hacen *new*'s ni *delete*'s porque ya se han hecho allí.

En el Algoritmo 4.12 se muestra una versión que extrae el árbol de exploración, T_{dfs}, del recorrido en profundidad.

```
void dfs(grafo_listas& g, int s, int p[])
{
    C[s] = gris; abro(s);
    para_todo_vecino(l,g[s]) {
        int v = l→v;
        if (C[v] == blanco) {
            p[v] = u;
            dfs(g,v);
    }
    C[s] = negro; cierro(s);
}
```

Algoritmo 4.12 *Exploración en profundidad con obtención del árbol* T_{dfs}.

Y así como en el caso de la exploración en anchura era frecuente la solicitud del vector de distancias a la raíz de la exploración, para el caso del recorrido en profundidad es frecuente solicitar dos vectores de tiempos. Un vector, a, que nos marque el tiempo de apertura de cada vértice, y otro, c, para el cierre. Estos tiempos no seran más que un entero que se incrementará cada vez que se haga una de las dos acciones sobre cualquier nodo. Para disponer de estas

informaciones, se debería modificar el Algoritmo 4.12. La nueva cabecera sería

void dfs(grafo_ listas& g, int u, int p[], int o[], int t[]).

Además deberíamos tener un contador declarado en alguna área de visibilidad global. Llamémosle tt. Lo inicializamos a cero en el Algoritmo 4.7 que llama al Algoritmo 4.12. Y finalmente, dentro de este último algoritmo, además de la modificación en la cabecera, deberíamos añadir

$$a[s] \; = \; tt; \; tt = tt + 1;$$

justo después de la tercera línea, bajo $C[s] \; = \; gris$; $abro(s)$; para tener los tiempos de apertura de todos los nodos.

Y también

$$c[s] \; = \; tt; \; tt = tt + 1;$$

justo después de la línea penúltima, donde dice $C[s] \; = \; negro$; $cierro(s)$; para disponer de los tiempos de cierre de cada nodo.

Aplicando el procedimiento del Algoritmo 4.12 al grafo no dirigido de la Figura 4.1 obtenemos un árbol como el de la Figura 4.12*(a)*. Los vectores de la Figura 4.12*(b)* contienen los valores de salida. El vector p que representa el árbol de exploración, el vector a que nos indica el orden de apertura de los vértices, comenzando en cero, y el vector c que nos dice el orden de cierre.

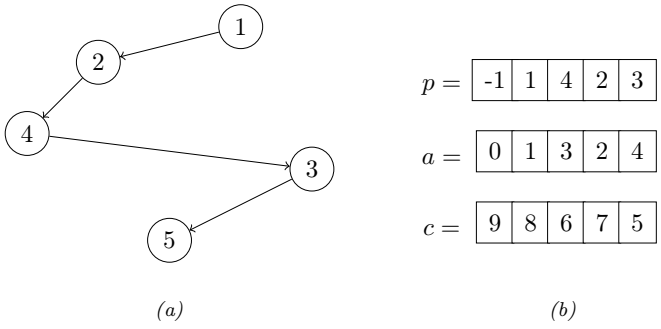


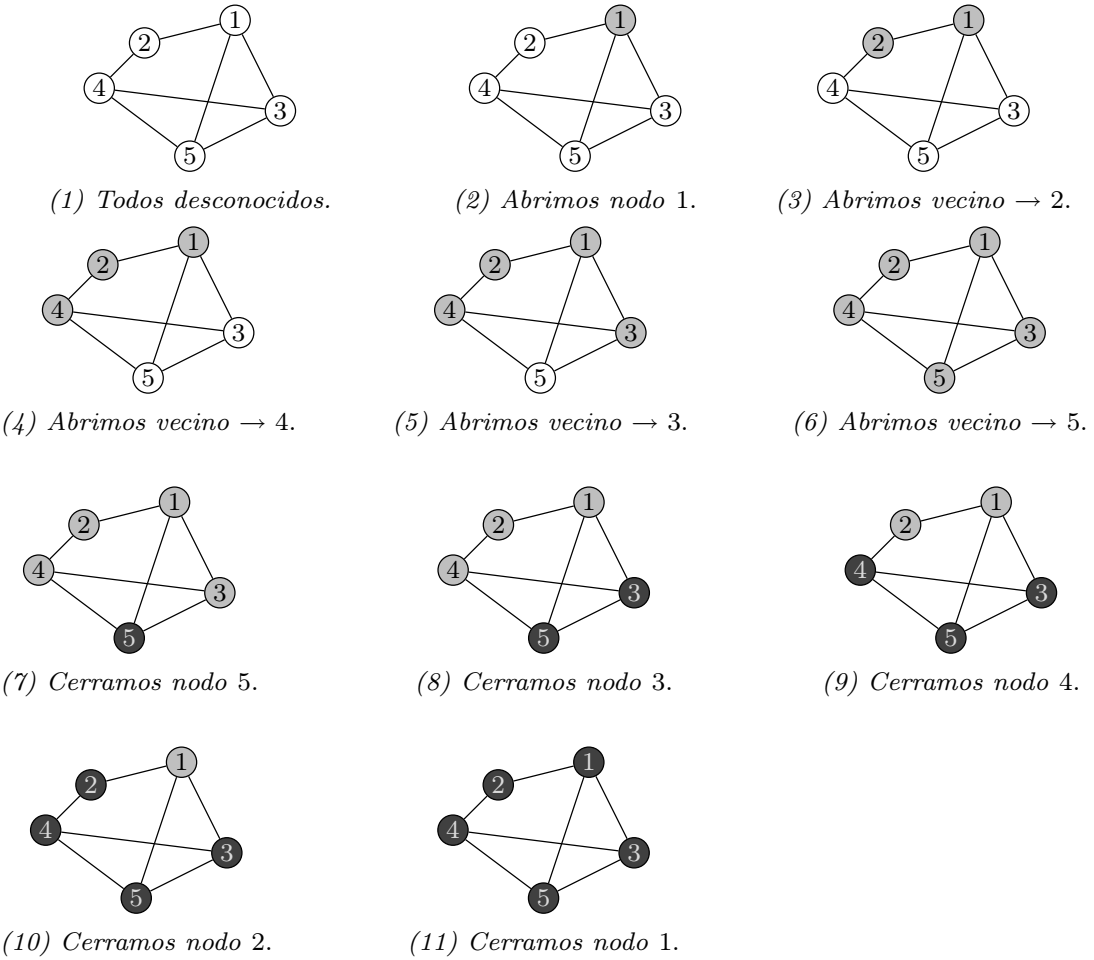
Figura 4.12: *(a) Árbol de exploración en profundidad, $T_{dfs}(G)$, para el grafo de la Figura 4.1.; (b) Valores de los vectores de salida.*

Este ejemplo es quizás demasiado sencillo, ya que el árbol de exploración es tan solo una lista. Sin embargo, si no fuera así pasaría que los valores de apertura y cierre estarían enredados. Es decir, con un ejemplo más complicado, habrían aparecido nodos cerrados antes que otros fueran abiertos.

Si todavía no habéis salido del castillo y estáis volviendo todo el rato al

recibidor, probablemente lo tengáis más que aburrido. Os aconsejo que olvidéis vuestra estrategia. A partir de ahora, cuando entremos en la próxima habitación, siguiente vecina de donde estéis, id mas lejos. Tanto como sea posible. Ciertamente, el recorrido en profundidad es más divertido.

En la Figura 4.13 se puede ver la evolución de los colores de los vértices para el mismo grafo de ejemplo durante todo el recorrido del Algoritmo 4.12. La asociación entre las líneas de código y las figuras, igualmente, no está demasiado clara. Efectivamente se puede relacionar la Figura 4.13*(1)* al estado inicial antes de la tercera línia. Es bien cierto asimismo que la Figura 4.13*(2)* es justo después de esta línea. Pero, en adelante, las próximas figuras son ya iteraciones del bucle principal.



(1) Todos desconocidos. *(2) Abrimos nodo* 1. *(3) Abrimos vecino* → 2.

(4) Abrimos vecino → 4. *(5) Abrimos vecino* → 3. *(6) Abrimos vecino* → 5.

(7) Cerramos nodo 5. *(8) Cerramos nodo* 3. *(9) Cerramos nodo* 4.

(10) Cerramos nodo 2. *(11) Cerramos nodo* 1.

Figura 4.13: *Evolución de marcas en los nodos de un recorrido en profundidad.*

Respeto al análisis de eficiencia para la exploración en profundidad, no hay nada de nuevo. Todo el razonamiento hecho para el algoritmo de recorrido en anchura es totalmente válido para el recorrido en profundidad.

La eficiencia de un recorrido en profundidad también es $\Theta(V + E)$.

Ordenación Topológica

El algoritmo de recorrido en profundidad de un grafo es utilizado masivamente en muchas aplicaciones del mercado. Entre otros usos, se utiliza para conocer las componentes conexas de un grafo, que es lo primero que se hace cuando no se sabe por cierto si un grafo es conexo. Además, la exploración en profundidad tiene muchas otras utilidades. Como ejemplo, veremos una aplicación muy interesante. Se trata de la *ordenación topológica*, una operación que se usa sólo en grafos dirigidos acíclicos.

Los grafos dirigidos acíclicos son un tipo de grafo especialmente frecuente. A menudo se les nombra con las siglas *dag*. Entre otros ámbitos, aparecen como grafos que sirven para expresar precedencias entre acciones en el tiempo. Es decir, grafos donde cada nodo representa una acción, y un arco $a = (u, v)$ que va de un nodo u a un nodo v, significa que la acción u tiene que ser efectuada antes que la acción v.

Una ordenación topológica de un grafo dirigido acíclico, $D = (N, A)$ es una ordenación secuencial de todos sus vértices de manera que si D contiene el arco (u, v), o sea, si $(u, v) \in A$, entonces u aparece antes que v en la ordenación, siendo $u, v \in N$.

El problema que pretendemos resolver, pues, es: Dado un grafo dirigido acíclico, obtener una ordenación topológica.

```
void ordenacion_topologica(grafo_listas& g, int u, lista& Q)
{
    C[u] = gris; abro(u);
    para_todo_vecino(l,g[u]) {
        int v = l→v;
        if (C[v] == blanco) dfs(g,v);
    }
    C[u] = negro; cierro(u);
    Q.anadir(u); // añade al principio de la lista
}
```

Algoritmo 4.13 *Ordenación Topológica.*

En el Algoritmo 4.13 podemos observar la similitud tan grande que hay entre la ordenación topólogica y la exploración en profundidad. En otras palabras, si

tenemos un grafo de precedencias, tan solo hay que recorrerlo en profundidad, y el orden inverso de cierre de los nodos nos dará una ordenación topológica. O sea, tan solo hay que añadir una pila al recorrido en profundidad. Cuando cerramos un nodo, lo ponemos en la pila. Cuando acabamos el recorrido, extrayendo todos los valores de la pila nos saldrán en ordenación topológica. El Algoritmo 4.13 utiliza una lista del mismo tipo que se ha definido para las listas de adyacencia en el Algoritmo 4.3.

El funcionamiento de todo el proceso se explica con un ejemplo extraído del libro de R. Sedgewick [22]. Se trata de analizar el proceso habitual de vestirse para un ser humano de género masculino. Un hombre. El grafo que podríamos obtener, fruto del análisis, podría tener el aspecto que se muestra en la Figura 4.14.

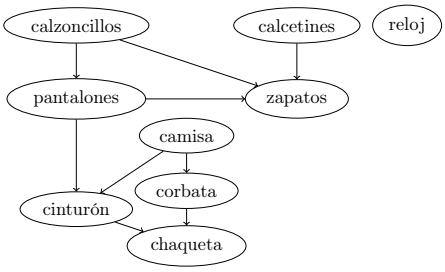


Figura 4.14: *Grafo de precedencias para el ejemplo de la ordenación topológica.*

A partir de este grafo de precedencias, el problema se trata de ordenar los vértices de manera que resulte alguna manera posible de que un hombre se vista. Invocamos el Algoritmo 4.13 con el grafo de la Figura 4.14.

Comenzaríamos por cualquiera de los vértices del grafo. Por ejemplo, por *camisa*. En la Figura 4.15*(1)* se muestra el estado correspondiente a la exploración recién iniciada. Todos los vértices son desconocidos, blancos, excepto el vértice *camisa* que ya ha sido abierto. La pila, Q, está vacía. Continuamos por la Figura 4.15*(2)* donde se puede ver que, según algún orden prestablecido, el primer sucesor del vértice *camisa*, *corbata*, también ha sido abierto. Procedemos con el primer sucesor de *corbata*, *chaqueta*, Figura 4.15*(3)*. Una vez abierto el vértice *chaqueta*, comprobamos que no tiene ningún sucesor que pueda abrir. Así pues, cerramos el vértice *chaqueta*, en la Figura 4.15*(4)*. En el momento en que cualquier vértice es cerrado pasa a encabezar la pila Q. Por esta razón, ahora Q = *(chaqueta)*. Cerrado el vértice *chaqueta*, retrocedemos a su padre en el árbol de exploración, el vértice *corbata*. Si pudiésemos, seguiríamos ahora por el siguiente sucesor desconocido. O sea en blanco. Pero no tiene más. Por tanto, cerramos el nodo *corbata*, Figura 4.15*(5)*, lo cual significa que también lo ponemos en la pila, quedando en la posición inicial. Ahora, Q = *(corbata, chaqueta)*. Continuamos. Como el nodo *corbata* ha quedado cerrado retrocedemos a su padre en el árbol, *camisa*, y buscamos el siguiente sucesor desconocido. Existe. Es el nodo *cinturón*. Lo abrimos, Figura 4.15*(6)*. Una vez abierto el vértice *cinturón*, no podemos abrir otros vértices.

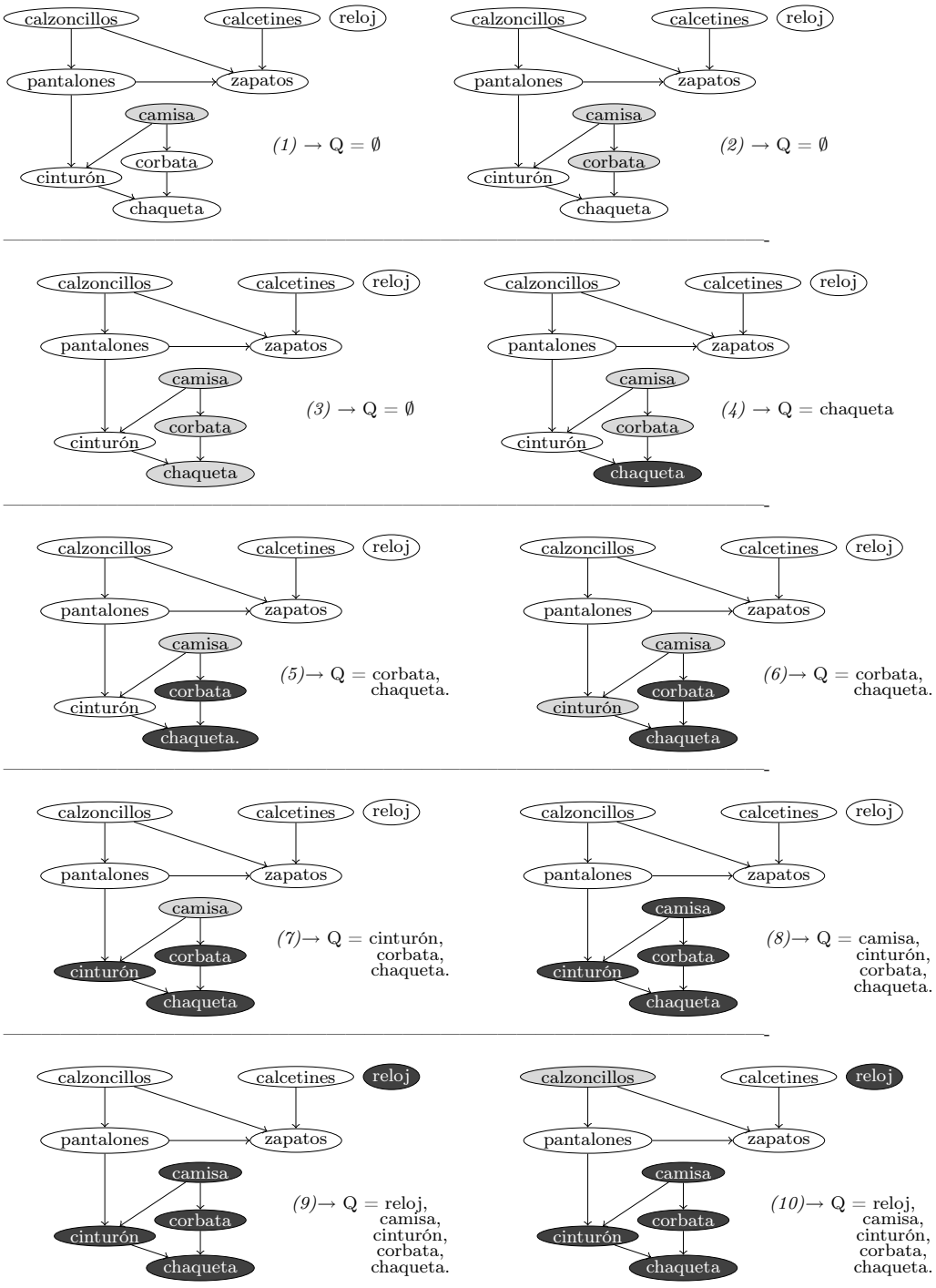


Figura 4.15: *Etapas iniciales del recorrido en profundidad.*

Por tanto, cerramos *cinturón*, Figura 4.15*(7)*. Ahora, *Q = (cinturón, corbata, chaqueta)*. Hemos cerrado el nodo, así que volvemos al padre, *camisa*. El vértice *camisa* ya no tiene más sucesores para abrir. O sea, cerramos *camisa*, Figura 4.15*(8)*. Tenemos *Q = (camisa, cinturón, corbata, chaqueta)*. Volvemos por primera vez del procedimiento del Algoritmo 4.13. Ahora bien, el módulo que lo ha llamado lo ha hecho desde un bucle que verifica que todos los nodos del grafo hayan sido cerrados. Y no es el caso. Entonces se vuelve a llamar a la función de ordenación topológica con un otro nodo raíz. Por ejemplo, podría ser el vértice *reloj*. Lanzamos una nueva exploración a partir del vértice *reloj*, y se acaba en seguida, ya que no tiene sucesores por abrir, y entonces se cierra inmediatamente después de abrirse. Y por tanto se añade a la pila por delante. Queda *Q = (reloj, camisa, cinturón, corbata, chaqueta)*, Figura 4.15*(9)*. Retornamos otra vez de la rutina, y como todavía quedan vértices en blanco, iniciamos un nuevo recorrido en profundidad a partir del vértice *calzoncillos*, por ejemplo. Por tanto, lo abrimos, Figura 4.15*(10)*.

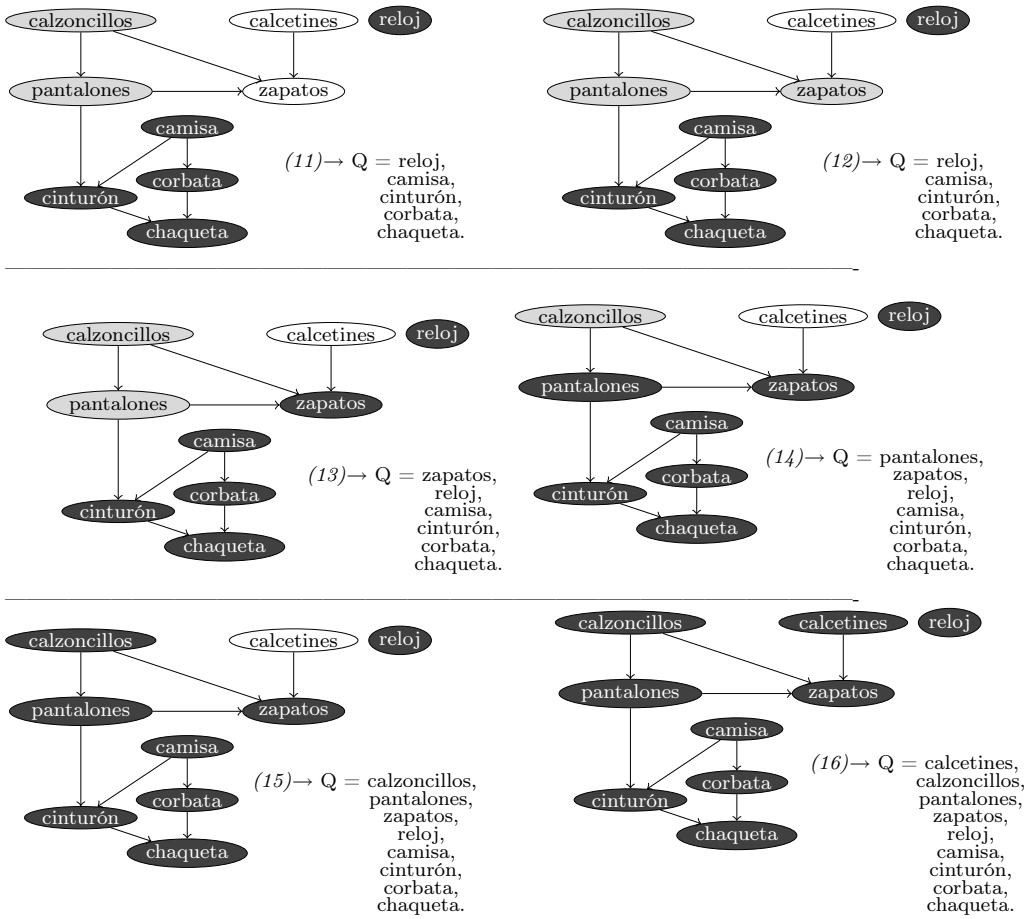


Figura 4.16: *Etapas finales del recorrido en profundidad.*

Y así seguimos abriendo sucesores mientras se pueda, cerrando vértices

cuando no se pueda abrir sucesores y añadiendo a la pila por delante, para retornar al padre y continuar con otros sucesores si los hay, Figuras 4.15*(11)* a la 4.15*(15)*.

Como se puede ver en la Figura 4.15*(17)*, a partir del grafo de precedencias hemos obtenido un orden para los nodos que respeta las precedencias que el grafo impone. La solución del problema da un orden en que un hombre puede vestirse. Es calcetines, calzoncillos, pantalones, zapatos, reloj, camisa, cinturón, corbata y chaqueta.

4.4 Grafos con Pesos

La experiencia nos enseña que los problemas más difíciles de resolver tienen grandes cantidades de datos iniciales, datos de entrada. Incluso, con el mismo concepto de grafo ya estamos insinuando un volumen de datos significativo. Por eso a los grafos, a veces, se les nombra *croquis*, *esquemas*, *diagramas*, y otros conceptos. Todos ellos, términos que huelen a cierta complejidad.

Como ya se ha dicho en la introducción de este capítulo, poquito a poquito vamos acumulando modelos que nos servirán para enfrentarnos al tipo de problemas que todavía no ha resuelto nunca nadie. Los grafos son un buen fundamento. Vamos a hacerlo crecer un poco. Entra un nuevo actor en escena. Una función de *pesos* asociada a las aristas del grafo.

El nombre que le damos a la función, *peso*, tiene que ser concebido de la manera más genérica. En el campo de la investigación operativa, hay disciplinas como la combinatoria poliédrica que estudia problemas de enrutamiento, donde lo llaman *coste*. Dicen que tienen un coste asociado a cada arista que se materializa en el coste de atravesar la arista. Para estos problemas conviene imaginarse un mapa de carreteras. Otros, *capacidad*. También en el campo de la investigación operativa hay problemas para los cuales la función real definida para las aristas se denomina capacidad. Son problemas de flujos en redes. En estos, conviene imaginarse una red en la qué las aristas son tubos que conducen agua, y las capacidades, los diámetros de los tubos. Y también hay quien al peso asociado a las aristas le llama *distancia*. En un plano más filosófico, se puede observar que así como el peso, el coste o la capacidad son atributos intrínsecos de la entidad arista, utilizando el término distancia, estamos dejando descansar directamente la definición de la función sobre la definición de los vértices de la arista con la qué está asociada.

En esta sección, se propone en primera instancia una definición formal para el concepto de grafos con pesos, y seguidamente se muestra una implementación en la estructura de datos *grafo_con_pesos*.

4.4.1 Definición

Para agilidad de la lectura, en adelante consideraremos la cuestión utilizando la terminología de los grafos no dirigidos. No obstante lo que se dice, es válido también para los dirigidos.

Definición 4.2 Grafo con pesos. *Decimos que $G = G(V, E, w)$ es un grafo con pesos si $G(V, E)$ es un grafo y w una función real de pesos definida sobre E,* $w : \underset{e}{E} \to \underset{w(e)}{\mathbb{R}}$.

Utilizamos w para el peso, del inglés *weight*.

Esto nos abre las puertas a lo que sería una relajación de lo que hasta ahora era una condición binaria. La existencía de las aristas. Es frecuente en muchos problemas considerar infinito los pesos de las aristas inexistentes, o cero cuando se trata de capacidades. Es decir, podemos cuantificar la existencia de las aristas. No hay que ser demasiado atrevido para decir que toda la teoría de grafos que se ha dado en este capítulo se hubiese podido dar con grafos con pesos, con los pesos de todas las aristas igual a 1. Así pues, los grafos con pesos en las aristas son una generalización propia de los grafos. Propia en el sentido que hay problemas que podemos representar con grafos con pesos, que sin los pesos no podríamos hacerlo.

En la Figura 4.17 se muestra un grafo de ejemplo. El peso de cada arista es el valor próximo a su punto central.

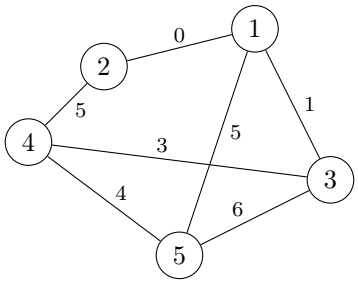


Figura 4.17: *Grafo no dirigido con pesos.*

En principio, los pesos pueden ser valores reales. Tocando de pies en el suelo encontraremos con planteamientos y soluciones de problemas que restringirán ese dominio a tan solo reales no negativos, o a reales estrictamente positivos.

4.4.2 Representación

De las dos implementaciones vistas para los grafos, la matriz de adyacencias nos resulta un pelo incómoda. Si restringimos la implementación a grafos simples, los que no tienen más de una arista entre cada par de nodos, entonces podemos hacer que la matriz de adyacencias sea de valores reales, *double*s, y directamente poner el valor del peso como contenido de la matriz. Deberíamos establecer un valor de peso para las aristas inexistentes, y podríamos aprovechar la implementación de la Sección 4.2.1 tan solo modificando los tipos de las variables. De todas formas, la rigidez de la estructura no nos permitiría ninguna otra modificación. Visto como van las cosas, no es difícil imaginar que en cualquier momento podríamos encontrarnos con problemas que requiriesen la definición de alguna función asociada a los vértices o de dos funciones diferentes para las aristas. Esos casos resultarían imposibles de implentar con la matriz de adyacencias. Ésa es una estructura monolítica que de tan compacta ya no erviría. Por eso, haremos la implementación de los grafos con pesos exclusivamente en listas de adyacencia, que es tan flexible como se necesite a la hora de añadir nuevos datos.

En la Figura 4.18*(a)* se puede observar el mismo grafo que en la Figura 4.17, y en la Figura 4.18*(b)* una representación gráfica del mismo grafo implementado en un vector de listas de adyacencia. Poner el primer decimal en los pesos de la estructura es para expresar que se trata de valores reales.

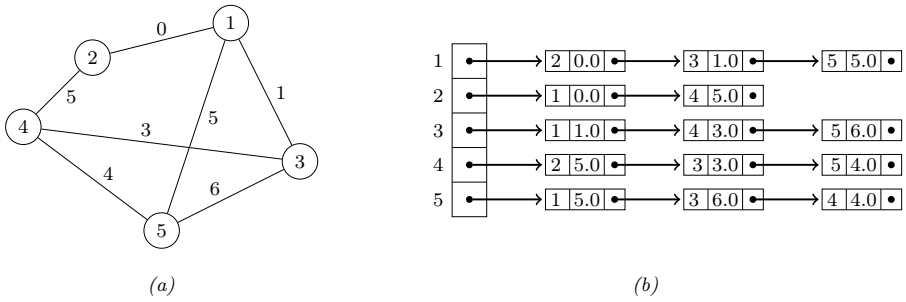


Figura 4.18: *(a) Grafo con pesos; (b) Representación en listas de adyacencia.*

Así pues, la implementación de los grafos en aristas resultará tan sencilla como hacer ligeras modificaciones a la implementación en listas vista en la Sección 4.2.2.

Comenzamos con los nodos que forman las listas. La estructura *arista* del Algoritmo 4.14 juega el rol que realizaba la estructura *nodo* en la Sección 4.2.2. Se corresponde con las cajitas de la Figura 4.18*(b)*.

La única diferencia con la implementación para los grafos sin pesos es precisamente la adición de la nueva variable miembro w, de tipo *double*.

```
struct arista {
    int v;
    double w;
    arista* siguiente;
    arista (int u, double peso)  v = u; w = peso; siguiente = NULL;
};
```

Algoritmo 4.14 *Estructura arista para grafos con pesos.*

Todos los cambios que la adición de esta nueva variable miembro provoque, son los que deberemos realizar para obtener la estructura para los grafos con pesos que andamos buscando.

```
struct lista_aristas {
    int n;
    arista* primera;

    void crea() {
        n = 0;
        primera = NULL;
    }

    void destruye(arista* p) {
        if (p && p→siguiente) destruye(p→siguiente);
        delete p;
    }

    void destruye() { destruye(primera); }

    void pon(int v, double w) {
        arista* nueva = new arista(v,w);
        nueva→siguiente = primera;
        primera = nueva;
        n++;
    }

    void quita() {
        arista* antigua = primera;
        primera = primera→siguiente;
        delete antigua;
        n--;
    }
};
```

Algoritmo 4.15 *Estructura* lista_aristas *para grafos con pesos.*

Como el análisis de esa estructura es del todo análogo al de la Sección 4.2.2, aquí tan solo se presenta la implementación.

En el Algoritmo 4.15 se describe una nueva definición de la estructura para las listas de adyacencia que formarán el grafo. Quizás convenga insistir en que se presenta un código tan pelado como es posible. Por esta razón la estructura mostrada en el Algoritmo 4.15, igual que en el caso de la Sección 4.2.2, sigue sin tener constructor ni destructor.

En pro de la usabilidad, no estaría de más que fueran implementados. En cualquier caso, no se van a utilizar en los algoritmos posteriores, y por otra parte, se asume que el lector no tendría inconveniente en implementar el código para ellas. En sintesis, serían un par de envoltorios públicos, *lista_ aristas()* y *∼lista_ aristas()* que llamarían a las funciones privadas que sí están implementadas en la estructura del Algoritmo 4.15.

También se puede observar que estas listas estan implementadas en forma de pilas por cuestiones de legibilidad. El aspecto que adquiere el código es realmente simple. Por otra parte, la eficiencia de cualquiera de las funciones de esa estructura es $\Theta(1)$.

Seguidamente, en el Algoritmo 4.16, se muestra la definición de la clase. Se puede pensar no sin razón, que utilizando plantillas en las definiciones de las estructuras podríamos haber parametrizado el código evitándonos esta nueva clase. En cualquier caso, valga el planteamiento hecho para la simplicidad y la legibilidad del código.

La única cosa que hace la clase *grafo_ pesos* mostrada en el Algoritmo 4.16 que no hiciese la clase *grafo_ listas* del Algoritmo 4.4 es la gestión de los pesos de las aristas. Sencillamente los tiene. Y basta.

```
class grafo_pesos {
    int n;
    lista_aristas* vector;
    bool dirigido;
    void crea() {vector = NULL; n=0; dirigido=false; }
    void crea(bool d) {crea(); dirigido = d; }
    void destruye() {
        for (int i=1; i<=n; i++) vector[i].destruye();
        delete [] vector;
    }

public:
    grafo_pesos(bool d) { crea(d); }
    ~grafo_pesos() { destruye(); }

    void anadir_vertice() {
        lista_aristas* aux = new lista_aristas[1+n+1];
        for (int i=1; i<=n; i++) aux[i] = vector[i];
        delete [] vector;
        n++;
        aux[n].crea();
        vector = aux;
    }

    void anadir_arista(int u, int v, double w) {          // 1≤ u,v≤n
        vector[u].pon(v,w);                               // no dirigido→ u>v
        if (!dirigido && u>v) anadir_arista(v,u,w);
    }

    lista_aristas& operator[](int i) { return vector[i]; }
    int tamano() { return n; }

    bool hay_arista(int u, int v) {
        para_todo_vecino(l,vector[u]) if (l→v == v) return true;
        return false;
    }
};
```

Algoritmo 4.16 *Declaración de la clase para la implementación de un grafo con pesos en listas de adyacencia.*

En este capítulo se ha hablado de grafos. Se han mostrado las dos representaciones más habituales, y los recorridos como operaciones paradigmáticas. También se ha visto que estas funciones requieren $\Theta(V + E)$, cosa previsible, ya que visitar un grafo significa visitar cada una de las aristas o cada uno de los nodos. Se ha puesto énfasis en los adjetivos de grafos implícitos y explícitos, insinuando que la idea va más allá que las representaciones materiales que podamos definir.

Finalmente, se ha establecido un fundamento para poder trabajar, en adelante, con grafos con pesos en las aristas.

Capítulo 5

Algoritmos Voraces

Los problemas con los que nos enfrentaremos en adelante son problemas de optimitzación. Optimitzar significa maximizar o minimizar. Lo sabe todo el mundo, que obtener máximos y mínimos es muy fácil. Se iguala la derivada de la función a cero y se resuelve la ecuación correspondiente. Y entonces, ¿Con qué problema nos vamos a enfrentar?

En el ámbito del análisis matemático se estudian funciones reales de variables reales. La característica más importante de las variables reales es la completitud. Eso equivale a decir que entre dos valores que pueda tomar la variable, puede tomar cualquiera de intermedio. Recordad que una variable independiente es aquélla a la que le podemos asignar el valor que convenga de su dominio, y que el dominio es el conjunto de valores que puede tomar una variable. El valor que le demos tendrá como consecuencia el valor que resulte de la función. Gracias a la completitud de las variables reales, podemos estudiar como responde la función al variar infinitamente poco el valor de las variables independientes. Es decir, podemos *derivar* la función respeto la variable independiente. Llegados a este punto, no hay que ser muy listo para utilizar el método del gradiente.

Una mañana cualquiera de invierno os ducháis. No queréis pasar frío en la ducha, así que empezáis poniendo el agua tan caliente como permita el grifo. Ya tenemos una función continua. La temperatura del agua depende del ángulo en que ponéis el grifo. Bien, esto una vez estabilizada. Pero, para conseguir estabilizar la temperatura, si sale demasiado caliente, cambiáis la posición del grifo para que salga más fría. Y si os pasáis, precisamente lo sabéis porque el agua sale demasiado fría. Y volvéis a probar a calentarla. En todo momento, el mismo error que obtenéis (el hecho de que el agua no salga a la temperatura óptima) os sirve de guía para saber lo que tenéis que hacer seguidamente. Esto es el método del gradiente. Ir en contra de la variación de la función con modificaciones cada vez menores de la variable independiente. Es un método tan popular que tiene diversos nombres. Cuando se estudian arquitecturas basadas en redes neuronales, por ejemplo, se llama contrapropagación del error. Las

redes neuronales que funcionan por contrapropagación del error procuran que el sesgo entre salida esperada y obtenida modifique el comportamiento de la red en la dirección correcta. A este proceso se le denomina aprenendizaje, o *training* en inglés.

Sin embargo, con todo, tanto de la operación de la derivada como del método del gradiente, no podemos sacar ningún provecho cuando las variables independientes son enteras. No podemos derivar. Ahora hacen falta nuevos métodos. Si cambio una calle de mi camino, normalmente tendré que cambiar más de una. No hay cambios infinitamente pequeños en una ordenación. Para cambiar un elemento, se cambian dos.

Este capítulo empieza con todo aquéllo que también es válido para los próximos capítulos, los problemas de optimización. Se introducen los problemas con variables enteras y se enuncia el principio de optimalidad. Se profundiza en los contenidos específicos del esquema algorítmico de los algoritmos voraces, en inglés *greedy algorithms*. Se presenta entonces el problema de la mochila, más relacionado con la programación dinámica que con los algoritmos voraces. Presentarlo anticipadamente ilustrará alguna limitación de la técnica algorítmica que en este capítulo se muestra. Y entonces se explican cuatro problemas más. El primero, *las gasolineras*, se ha escogido como uno de los paradigmas para mostrar la utilidad del esquema en el aspecto más sencillo. Los otros tres, Dijkstra, Kruskal y Jarník (Prim), son problemas que se plantean sobre grafos con pesos. Ejercitaremos pues gran parte de las estructuras vistas en capítulos anteriores.

5.1 Problemas de Optimización Combinatoria

Tanto éste como los dos próximos capítulos tratan de algoritmos focalizados en problemas de optimización de funciones de variables enteras, o discretas.

En principio, utilizamos el relativo *de decisión* para referirnos a variables binarias. O sea, que pueden tomar los valores cierto o falso, cero o uno, sí o no, blanco o negro..., o incluso, ser o no ser, que también es un dilema, [24]. Entonces las llamamos variables de decisión. Y también llamamos problemas de decisión aquellos problemas que la solución final es sí, o no. Pero claro, maximizar cosas que sólo pueden valer cero o uno suena extraño. Por eso, los problemas de decisión no son incluidos dentro de los de optimización. Estos problemas se verán en detalle en el Capítulo 8. Aunque las variables de decisión por excelencia sean las variables binarias, también llamamos variables de decisión a las variables enteras en general, flexibilizando la definición inicial. Para el caso de los problemas, en cambio, se usa el término con todo el rigor. Los problemas decisionales concluyen en cierto, o falso.

Observad que cuando trabajamos con variables binarias, estamos trabajando

con variables enteras. Es decir, el cero y el uno son números enteros.

Así pues, los problemas de optimización combinatoria son los que buscan puntos extremos a funciones lineales del tipo

$$f : \underset{x}{\mathbb{Z}^n} \to \underset{f(x)}{\mathbb{R}} \tag{5.1}$$

ya sean máximos o mínimos. A esa función le llamamos objetivo del problema. Pero claro, al ser lineales, son funciones monótonas. O sea, que si estos problemas se planteasen únicamente con funciones como la de la expresión (5.1), la solución quedaría siempre indefinida a $+\infty$ o $-\infty$. Conclusión, son necesarios más datos para refinar el problema. Y efectivamente así es. Además de la función donde buscar puntos singulares, los problemas de optimización combinatoria restringen el dominio de la variable independiente a una región $P \subset \mathbb{Z}^n$. En otras palabras, para cada vector de componentes enteras perteneciente a una región $x \in P \subset \mathbb{Z}^n$, que representa las variables independientes $x = (x_1, x_2, \ldots, x_n)$, la función a optimizar da un valor asociado $f = f(x)$, o si se prefiere $f = f(x_1, x_2, \ldots, x_n)$.

Tenemos n decisiones a tomar para optimizar el valor de una función. Fácil, probamos todas las combinaciones y miramos para cuál da el valor óptimo.

Fijaos bien en el contraste. Las variables continuas pueden ser modificadas infinitamente poco, cosa que permite encontrar puntos singulares en las funciones de variable continua, pero en cambio, no pueden tomar valores por enumeración. Y a la inversa, las variables enteras pueden tomar valores por enumeración, cosa que nos permite encontrar puntos singulares a funciones de variable discreta, pero no pueden ser modificadas infinitamente poco.

No obstante, solucionar problemas por enumeración no es tan viable como en el caso continuo derivar. Eso es así porque, tan solo tratándose de decisiones binarias serán 2^n posibles combinaciones, y si en lugar de binarias fueran variables enteras generales, peor todavía. Se nos va de madre, $\Omega(2^n)$. Tendremos que buscar otros métodos.

Bien, abramos un paréntesis para ver si conseguimos aclarar una confusión muy común. Lo que sigue es difícil de entender, pero también muy esclarecedor. Así pues, que quede claro. Es importante para hablar con propiedad sobre esta materia:

El valor óptimo de una función no tiene por qué poderse obtener de forma única.

Ignorar esto es un error grave que demuestra poca familiaridad con el tema por parte de la gente que lo comete. Cuando hablamos de un vector de valores concretos x^* que minimiza o maximiza f, casi siempre hay que decir que se busca *un* valor óptimo $x^* = (x_1^*, x_2^*, \ldots, x_n^*)$. Muy excepcionalmente deberemos decir *el* valor óptimo cuando nos referimos al valor de las variables que provocan el máximo o el mínimo en la función. Siempre *un*. O *algún*. Y casi nunca *el*. De hecho, tanto es así, que si sólo hay un valor de x, sea x^*, para el cual f es

óptima, entonces merece la pena insistir. Decir *el único* óptimo de f es $x = x^*$. En cambio, cuando nos referimos al valor resultante $f^* = f(x^*)$, entonces sí que hay que decir siempre *el* valor óptimo. Claro, el hecho que las dos cosas, x^* y f^*, estén tan ligadas hace que a menudo se confundan los términos... y ya cerramos el paréntesis.

Más cosas. Solucionar un problema de optimización combinatoria significa dar el valor real f^*. El valor óptimo de la función. Todo esto está relacionado con la segunda mitad de la frase que abre este libro, *qué es a cómo lo que cuántos es a cuáles*. Solucionar el problema es decir cuántos, y si se quiere, además, cuáles.

Ya se ha mencionado que el vínculo entre x^* y f^* no tiene por qué ser bidireccional, f puede ser epiyectiva. Que yo sepa qué distancia tiene el camino más corto entre dos pueblos no significa que sepa ir de uno al otro. Por otro lado, si nos dan un valor concreto x^* podemos calcular en poco tiempo qué valor de la función objetivo f^* le corresponde, $f^* = f(x^*)$. O sea, si sé ir por uno de los caminos más cortos, puedo medirlo en poco tiempo y saber la distancia más corta entre los dos pueblos, que es el valor de la función objetivo.

Ir por un camino significa tomar una decisión en cada bifurcación. El conjunto de decisiones son el problema. Entre ambos pueblos, el camino más corto medirá una cierta distancia, pero puede no ser un único camino. Cada uno de los caminos más cortos (todos ellos de igual distancia) puede pasar por diferentes lugares. Cuando ése es el caso, el método de solución toma protagonismo, ya que si el valor óptimo de la función objetivo se produce en diferentes óptimos para los valores de las variables, entonces cualquiera de las soluciones posibles es igualmente válida, y la proporcionada con el método en cuestión acostumbra a ser fruto de ordenaciones implícitas de los distintos conjuntos no necesariamente ordenados que intervienen.

Eso trasciende a los algoritmos. Para no sólo saber $f^* = f(x^*)$ sinó también los valores concretos de las n variables de x^*, o sea, para saber cuáles además de cuántos, será necesario añadir estructuras de datos específicas.

5.1.1 Factibilidad

Tal como se ha definido la función a optimizar, los problemas de optimización combinatoria pueden expresarse como

$$\begin{aligned} max\ f : \mathbb{Z}^n &\rightarrow \mathbb{R} \\ x \in P, P &\subset \mathbb{Z}^n. \end{aligned} \tag{5.2}$$

A la región P incluida en el espacio de los vectores de n variables enteras le denominamos *región de factibilidad*.

Para problemas de optimización con variables de decisión, o también enteras generales, se utiliza profusamente la Programación Lineal, disciplina fundamental de la Programación Matemática. En este ámbito, la región de factibilidad es un poliedro. Eso significa que es un espacio definido por un sistema de desigualdades en las que nunca aparecen productos de variables. Geométricamente pues, resolver estos problemas consiste inicialmente en delimitar un espacio formado por planos que intersectan esculpiendo un poliedro. De esta forma, nos encontramos con formulaciones como

$$\text{maximizar} \qquad c_1x_1 + c_2x_2 + \cdots + c_nx_n \tag{5.3}$$

$$\begin{aligned} \text{satisfaciendo} \qquad & a_{1,1}x_1 + a_{1,2}x_2 + \ldots + a_{1,n}x_n \leq b_1 \\ & a_{2,1}x_1 + a_{2,2}x_2 + \ldots + a_{2,n}x_n \leq b_2 \\ & \ldots \qquad \ldots \qquad \ldots \\ & a_{m,1}x_1 + a_{n,2}x_2 + \ldots + a_{m,n}x_n \leq b_m \end{aligned} \tag{5.4}$$

donde tanto c_i, para $i = 1, \ldots, n$, como $a_{i,j}$, para $i = 1, \ldots, m$ y $j = 1, \ldots, n$, como los b_i, $i = 1, \ldots, m$, son números reales.

La expresión (5.3) es la función objetivo. El conjunto de desigualdades (5.4) es el conjunto de planos que limitan el poliedro de factibilidad.

En notación vectorial, si tenemos $c \in \mathbb{R}^n$, $A \in \mathbb{R}^{n \times m}$, y $b \in \mathbb{R}^m$, entonces el problema es

$$\text{maximizar} \qquad c^T x \tag{5.5}$$

$$\text{satisfaciendo} \qquad Ax \leq b \tag{5.6}$$

Cuando hablamos de un vector suponemos siempre que lo representamos como un vector columna. Por esta razón en la función objetivo (5.5) del problema tenemos el vector c transpuesto, c^T. Eso provoca que cuando dimensionamos una matriz digamos primero la dimensión en columnas. La expresión (5.6) es la matriz de restricciones del problema lineal. Esta matriz A tiene pues n columnas y m filas. A menudo ocurre que los estudiantes se sienten incómodos cuando, indexando una matriz, el primer índice corresponde a las columnas. En cambio, cuando se indexa un espacio continuo entonces acostumbran a poner primero la dimensión horizontal, denotando (x, y) a un punto del plano en lugar de (y, x). Y por tanto, poniendo primero lo que serían las columnas. Así pues, se trata de tener claro en cada momento qué es lo que se está diciendo, y no hacer caso de costumbres cuando nos incomodan.

Al conjunto de restricciones $Ax \leq b$, también se le denomina poliedro del problema o modelo poliédrico. El poliedro caracteriza el problema. El vínculo entre el modelo poliédrico y el problema es tan importante, que la disciplina que estudia estos problemas también es conocida como Combinatoria Poliédrica. Esto es así porque la complejidad de los problemas va ligada al número de planos

necesarios para caracterizar el poliedro, aparte del hecho de tener que producir soluciones enteras. Ocurre frecuentemente, para los problemas más complejos, que $m \in \Omega(2^n)$ imposibilitando el establecimiento de todas las filas de la matriz A a la hora de hacer la descripción poliédrica.

Describir estos planos es el quid de la programación matemática. Cuando se entiende, es muy divertida. Eso sí, no siempre es fácil. Por ejemplo, seguidamente se muestra la formulación de algunos de los planos que cerrarían el espacio de factibilidad de algún problema. Observad como haciendo la tarea que se hace en este ejemplo se respira el mismo aire que en la programación informática, ya que en definitiva, se está codifcando una lista de requerimientos del usuario. Se trata de un divertimento sin ningún rigor. No se enuncia ni siquiera la función objetivo. El propósito de estos próximos párrafos es tan solo aprender a traducir del lenguaje natural al lenguaje formal utilizado en programación matemática. Se aborda el ejemplo para el caso más sencillo, el de variables binarias. Hay libros [27], [21] que hablan de cómo sistematizar este tipo de traducciones, aunque no son sencillos de comprender y es necesario cierto conocimiento del lenguaje formal.

Imaginemos que, como inversores, tenemos diez posibles proyectos en los cuales podemos invertir. Las inversiones no pueden ser fraccionadas, hay que invertir en todo un proyecto o no invertir nada, pero no nos podemos quedar a medias tintas.

Para el planteamiento del problema definimos una variable de decisión para cada proyecto, x_i, para $i = \{1, \ldots, 10\}$. Si x_j vale 1 significa que invertiremos en el proyecto j. Y si x_j vale cero, pues que no.

Y ahora juguemos a formular restricciones inventadas sobre cuáles serán los proyectos escogidos. Sabemos que $x_i \in \{0, 1\},\ i = \{1, \ldots, 10\}$.

- Seguro que invertimos en el proyecto cuatro $\rightarrow x_4 = 1$.
- Sólo podemos invertir en el segundo si invertimos en el primero $\rightarrow x_2 \leq x_1$.
- Invertimos en el séptimo o en el noveno, pero no en ambos $\rightarrow x_7 + x_9 = 1$.
- Si invertimos en el octavo, entonces también en el tercero o el cuarto $\rightarrow x_8 \leq x_3 + x_4$.
- Como mínimo tenemos que invertir en tres proyectos $\rightarrow \sum_{i=1}^{10} x_i > 3$.
- Hay que invertir exactamente en un total de cinco proyectos $\rightarrow \sum_{i=1}^{10} x_i = 5$.
- No podemos invertir en los diez proyectos $\rightarrow \sum_{i=1}^{10} x_i < 10$.

Aunque en la definición del poliedro $Ax \leq b$ se utilicen tan solo operadores de $\leq$ podemos cambiar de signo las dos partes de las desigualdades obteniendo el operador $>$. También podemos utilizar $\geq$ si añadimos una unidad al término de la izquierda cuando tenemos el operador $>$. Además, también podemos sobreentender que el operador $=$ se puede obtener diciendo las dos cosas a la vez, $\leq$ y $\geq$.

Está claro que si en el mismo problema expresamos condiciones contradictorias, entonces la región de factibilidad será vacía. No podemos decir, por un lado que queremos invertir en más de tres proyectos, y por otro en menos de dos. Eso significa que los algoritmos tendrán una posible salida que dirá UNFEASIBLE SOLUTION.

5.1.2 Principio de Optimalidad

El principio de optimalidad de Bellman también conocido como *propiedad de subestructuras óptimas* dice así.

Proposición 5.1 Principio de Optimalidad. *Dada una secuencia óptima de decisiones que resuelve un problema, toda subsecuencia es óptima del subproblema que representa.*

El principio de optimalidad sirve para demostrar la optimalidad de un método por inducción.

El camino más corto entre dos puntos está formado por los caminos más cortos entre cualquier pareja de puntos intermedios. Ésta es una buena manera de recordar este principio. Los caminos mínimos están hechos de caminos mínimos.

Para el caso de los algoritmos voraces, tan solo utilizaremos este principio cuando tengamos que demostrar que un cierto algoritmo proporciona soluciones óptimas para las instancias de entrada, como ya se ha dicho más arriba. En el marco de la programación dinámica, sin duda, el uso de este principio es mucho más necesario. Toda la programación dinámica descansa sobre este principio. Como veremos en el Capítulo 6, la metodología para la construcción de las soluciones óptimas se fundamenta en el principio de optimalidad. Por ese motivo, esta sección podría haber sido ubicada legítimamente en el próximo capítulo. Está aquí para poder escribir la demostración del teorema de la Sección 5.4, del problema de las gasolineras. Finalmente, en los problemas de búsqueda exhaustiva, Capítulo 7, utilizaremos el principio de optimalidad para evitar cálculos. Si pretendemos probar todos los valores factibles del conjunto de variables al solucionar un problema, podemos utilizar este principio para no profundizar en la exploración de una solución si a medio camino ya vemos que la solución parcial no es óptima.

5.2 Esquema Algorítmico de Algoritmos Voraces

El método más sencillo para atacar problemas de optimización combinatoria es el de los algoritmos voraces, que se muestra en el Esquema 5.1. Con este algoritmo, a veces, obtendremos soluciones óptimas. No siempre. Depende de la naturaleza del problema.

Se parte de un conjunto de candidatos C, de los cuales un subconjunto S formarán la solución. La secuencia de decisiones será, para cada candidato decidir si forma parte de la solución, o no.

```
algoritmo voraz(C:conjunto) retorna S(conjunto)
{
    S ← ∅
    mientras no solución(S) y C ≠ ∅
        x ← seleccionar(C)
        C ← C \ {x}
        si factible(S ∪ {x}) entonces S ← S ∪ {x}
    fmientras
    retorna S
}
```

Esquema 5.1 *Algoritmos Voraces.*

Los tipos de problemas para los cuales este esquema algorítmico obtiene soluciones óptimas son tratados desde la programación lineal por medio de conceptos como *matroides* y cosas bien extrañas, referidas exclusivamente a propiedades de la matriz de restricciones que limita el poliedro de factibilidad.

Dejando aparte la teoría matemática que soporta este tipo de problemas, el aspecto más relevante del Esquema 5.1 es la incapacidad para preguntarse dos veces si un mismo candidato ha de formar parte de la solución. Eso es un inconveniente grave que limita mucho las posibilidades de ese esquema.

Se parte del espacio $\mathbb{Z}^n$. Eso viene dado por el conjunto C en el Esquema 5.1, que tiene cardinalidad $n = |C|$. Como se ha mencionado, a partir de este conjunto de decisiones, el esquema retorna un subconjunto $S \subset C$ de aquellas decisiones que se han tomado ciertamente.

Fisiológicamente, sobresalen propiedades características. La más importante es que el esquema nos muestra un bucle *while*, o *for*. En cada iteración se toma una decisión, caiga quien caiga. O sea, si fuera por este esquema, los algoritmos tardarían $\Theta(n)$. Pero no será tan fácil. Las operaciones internas no tienen por qué ser $\Theta(1)$. Y todavía más grave, antes del bucle, a parte de la inicialización

de la solución S, a menudo también se hacen ordenaciones de C, haciendo el código $\Omega(n\ log\ n)$.

Llamamos *selección voraz* a la selección del nuevo candidato, x en el Esquema 5.1. La selección voraz va asociada a la idea astuta que, cuanto mejor sea, mejor será el valor final obtenido. La función de selección caracteriza el algoritmo. No el problema. Es decir, un mismo problema puede ser implementado con diversos algoritmos voraces con diferentes criterios de selección voraz.

Es en este punto donde nos olvidamos de aquéllo dicho en el primer capítulo, página 19, referente a que asociaríamos los conceptos de problema y algoritmo uno a uno. En adelante consideraremos que para un mismo problema se pueden usar diversos algoritmos.

Cuando la naturaleza del problema es tal que se puede asegurar que se utiliza una selección voraz óptima, entonces la técnica de los algoritmos voraces nos proporciona soluciones óptimas, por el principio de optimalidad. Para que la solución voraz sea óptima las decisiones tienen que ser independientes, o al menos, tienen que poder ordenarse de manera que ninguna decisión dependa de otra posterior. Y para complicar más la cosa, a veces una misma selección voraz puede ser óptima o no dependiendo de los valores concretos de la instancia que se pretende resolver. Éste es el caso del problema de la mochila. Para este tipo de problemas no podemos decir que los algoritmos voraces den soluciones óptimas, ya que cuando lo decimos queremos decir para cualquier instancia.

Volvamos al Esquema 5.1. En cada iteración del bucle se reduce en 1 el valor del cardinal del conjunto de entrada, $|C|$. Eso significa que, una vez seleccionado, descartamos el nuevo candidato. Los candidatos dejan de serlo cuando los analizamos. Después ya veremos si los seleccionamos para la solución, pero de entrada, descartamos volverlos a analizar. Y para analizarlos, nos preguntamos si añadiendo este nuevo candidato en la solución saldríamos de la región de factibilidad. Es decir, si es posible aceptar el nuevo candidato. Y si lo es, lo aceptamos.

En definitiva, la calidad de la solución obtenida depende fuertemente de la ordenación del conjunto de candidatos, ya que la resultante es la primera solución factible de dimensión n que se encuentra. Las repercusiones de esta conclusión son claras. Como veremos en este capítulo, los algoritmos voraces acostumbran a empezar ordenando de alguna manera el conjunto de candidatos.

5.3 Problema de la Mochila

El problema de la mochila (*the knapsack problem*) es muy conocido. Desde los inicios de la investigación operativa y la programación lineal hay bibliografía específica para este problema, [14]. Y otra de más reciente, [17]. Ha sido uno de

los problemas emblemáticos de varias disciplinas. Fue un problema pionero de la programación lineal, y también lo ha sido en algoritmos genéticos [3], redes neuronales [26], o de colonias de hormigas [13]. Y también, sin duda, de la programación dinámica [6], [25], o [23].

Todo el mundo se ha planteado el problema de la mochila en algún momento. Al llenar el maletero del coche, o una caja de cartón en una mudanza. Dice así.

Definición 5.1 Problema de la Mochila. *De entre n objetos que tienen pesos $w_i \in \mathbb{R}$, para $i = \{1, \ldots, n\}$, y valores $v_i \in \mathbb{R}$, también para $i = \{1, \ldots, n\}$, conseguir el máximo valor posible, siempre que el peso total no supere la capacidad de peso W de la mochila.*

Formalmente, llamamos x a un vector de dimensión n. Cada componente x_i nos indica la cantidad de objetos del tipo i que meteremos en la mochila, para $i = \{1, \ldots, n\}$. Tenemos así que el problema puede ser expresado como, dados n pesos $w_i \in \mathbb{R}$ y n valores $v_i \in \mathbb{R}$, para $i = \{1, \ldots, n\}$,

$$\begin{array}{lll} \text{(MOCHILA)} & \text{maximizar} & v_1x_1 + v_2x_2 + \cdots + v_nx_n \\ & \text{satisfaciendo} & w_1x_1 + w_2x_2 + \cdots + w_nx_n \ \leq W \\ & & x_i \geq 0,\ x_i \in \mathbb{Z},\ i = \{1, \ldots, n\} \end{array} \tag{5.7}$$

En notación vectorial, si $v, w \in \mathbb{R}^n$, la formulación poliédrica del problema de la mochila se puede describir como sigue

$$\begin{array}{lll} \text{(MOCHILA)} & \text{maximizar} & v^Tx \\ & \text{satisfaciendo} & wx \leq W \\ & & x \geq 0,\ x \in \mathbb{Z}^n \end{array} \tag{5.8}$$

Así pues, se entiende que sea un problema fundamental, ya que es la versión más sencilla del problema genérico planteado con el modelo de las expresiones (5.3) y (5.4) de la página 203.

En el Algoritmo 5.1 se muestra una implementación voraz para el problema de la mochila. Se ha usado una cola de prioridad para mínimos en lugar de un quicksort para no añadir más código en este capítulo. El criterio de selección voraz consiste en ordenar los objetos según la relación peso valor. O sea, según el coeficiente peso dividido por valor. Así pues, los mejores objetos serán los que tengan ese valor más pequeño.

```
double mochilla(int n, double w[], double v[], double W)
{
      cola_de_prioridad_minheap Q(n);
      for (int i=0; i<n; i++) Q.insertar(i,w[i]/v[i]);
      double peso = 0.0;
      double valor = 0.0;

      while (!Q.vacia()) {
            item objeto = Q.getmin();
            int j = objeto.index;
            if (peso + w[j] < W) {
                  valor = valor + v[j];
                  peso = peso + w[j];
            }
      }
      return valor;
}
```

Algoritmo 5.1 *Algoritmo voraz para el problema de la mochila.*

Respeto a la optimalidad, este algoritmo da el óptimo según sean los valores de la instancia del problema. Si suponemos, por ejemplo, que todos los pesos son iguales, o que todos los valores son iguales, entonces efectivamente obtendríamos valores óptimos. Es fácil imaginarse instancias en las que el algoritmo voraz da soluciones óptimas del problema. No obstante, si tomamos por ejemplo una mochila con una capacidad $W = 3$, y $n = 2$ objetos con pesos $w_1 = 2$, y $w_2 = 3$, con valores $v_1 = 4$ y $v_2 = 5$, la solución valdría sólo 4, mientras el óptimo es 5. Esto es debido a que, al calcularse los ratios, nos encontramos con los índices ordenados $(1, 2)$, ya que $2/4 < 3/5$. De hecho, efectivamente es más rentable el objeto 1 que el 2. Lo que provoca que la solución no sea óptima, pues, es el valor de la capacidad, ya que si en lugar de 3 fuese 4, el Algoritmo 5.1 daría el óptimo para esta instancia.

Queda claro así, que la metodología voraz, con una selección basada en la ordenación de los cocientes peso valor, no garantiza el óptimo para el problema de la mochila.

Preguntarse sobre si podemos coger varios objetos del mismo tipo o no, no trasciende en el algoritmo voraz. Es lo mismo que preguntarse si la x debe ser binaria o entera. No tiene ninguna relevancia, ya que admitimos objetos repetidos en los datos. O sea, que pueden existir objetos i y j, tales que $w_i = w_j$ y $v_i = v_j$ siendo $i \neq j$.

La eficiencia del Algoritmo 5.1 viene dominada por la ordenación que se hace de los objetos antes del bucle, $\Theta(n \; log \; n)$.

5.4 El Ejemplo de las Gasolineras

Un coche tiene que hacer el trayecto de Sitges a Ondarroa. Sale de Sitges con el depósito lleno. Se dispone de un mapa con el recorrido que debe hacer el coche, las n gasolineras que hay en el trayecto y las distancias entre cada una de ellas. El coche, que es un tartana vieja, no puede hacer más de cien kilómetros sin repostar. El problema es minimizar el número de paradas que deberá hacer, satisfaciendo que la distancia entre paradas tiene que ser inferior a $D = 100$.

El algoritmo voraz que soluciona el problema de las gasolineras establece como selección voraz parar lo más tarde posible. No hace falta ningún gran esfuerzo para intuirlo. Saliendo de Sitges, no paramos hasta la gasolinera más lejana de las que están antes de 100 Km. Entonces, y hasta llegar a Ondarroa, repostaremos siempre en la que esté lo más lejos posible del lugar actual, pero a menos de 100 Km.

En el Algoritmo 5.2 se puede ver una implementación de este método utilizando la estrategia voraz.

```
int gasolineras(int n, double b[], double D, double p[])
{
    int i = 0;
    p[i] = 0.0;
    for (int j=0; j<n; j++) {
        if (b[j]-p[i] >= D) {
            i++;
            p[i] = b[j-1];
        }
    }
    return i;
}
```

Algoritmo 5.2 *Algoritmo voraz para el problema de las gasolineras.*

La resolución del problema es sencilla. A la rutina del Algoritmo 5.2 se le pasa el número n de gasolineras en la ruta, la ubicación de cada una de ellas respeto al punto inicial en kilómetros, b, la distancia máxima D que se puede hacer sin parar, y, como salida, nos retorna el número de paradas necesarias y las ubicaciones de cada una de ellas, p. No hay detección de solución no factible. La región de factibilidad sería vacía si hubiese dos gasolineras consecutivas a más de D kilómetros.

Esta sencilla idea se ilustra en la Figura 5.1.

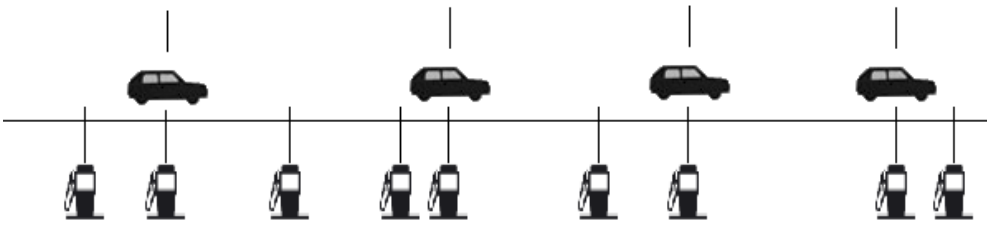


Figura 5.1: *Idea intuitiva para la selección voraz en el problema de las gasolineras.*

Respeto a la optimalidad, podemos formalizarla con el siguiente postulado.

Teorema 5.1 Optimalidad voraz en el problema de las gasolineras. *El Algoritmo 5.2 soluciona óptimamente el problema de las gasolineras.*

Prueba *Teniendo en cuenta que el problema satisface el Principio de Optimalidad, tan solo hay que demostrar que la selección voraz de posponer la próxima parada tanto como sea posible es una selección óptima.*

Para demostrar que la selección de la siguiente gasolinera es óptima, llamemos S a nuestra solución, y supongamos que existiese otra S' que fuese óptima con algún otro criterio de selección. Entonces, si $S'[1] \neq S[1]$, pasaría que $S'[1] < S[1]$, por definición de nuestro criterio de selección. Así pues, si a esta supuesta solución S' le cambiamos la primera decisión, obteniendo $S'' = S' \setminus \{S'[1]\} \cup \{S[1]\}$ no empeoraría, ya que es seguro que $S'[2]$ es inferior o igual a $S[1] + D$. O sea, que S', como mínimo, tendrá el mismo número de paradas que S. Y, por tanto, S también es óptima. □

El teorema 5.1 y su demostración tienen una estructura que es conveniente comprender y aprender. Se trata de demostrar la optimalidad de un problema resuelto con una estrategia voraz. El primer paso que hace la demostración es recordar el principio de subestructuras óptimas para justificar que tan solo hay que demostrar la optimalidad de la selección voraz. Lógicamente, el segundo paso es la demostración de que la selección voraz es óptima. Esto se hace introduciendo una solución, que se supone óptima, obtenida con un criterio de selección diferente del actual, y comprobando que si en la nueva solución se le cambia la primera decisión por la decisión de nuestro algoritmo, la solución no podría mejorar. Y por tanto, la nuestra es tan buena como la solución supuesta óptima. Con más rigor, en el momento de referirnos al primer elemento, deberíamos referirnos al primer elemento diferente. Y también concluir con la posibilidad de que no exista un primer elemento distinto. El hecho de no encontrar ninguno distinto hace que la solución supuestamente óptima sea la nuestra directamente. Siempre que nos pidan demostrar que un algoritmo voraz proporciona soluciones óptimas, utilizamos este esquema para la demostración.

Las eficiencias del Algoritmo 5.2, tanto temporal como espacial, está bien

claro que son $\Theta(n)$.

5.5 Caminos Mínimos

Los algoritmos que siguen en este capítulo hacen referencia a grafos con pesos. Todos ellos. Por eso, vaya por delante que cuando decimos grafos nos referimos a grafos con pesos, que también podemos llamar distancias o costes. En particular, aquí hablaremos del problema de caminos mínimos (en inglés *shortest paths*).

Definición 5.2 Problema de Caminos Mínimos. *Dado un grafo, $G(V, E)$, dos vértices, $s, t \in V$, y una función real de distancias asociada a las aristas, $d : E \to \mathbb{R}$, averiguar la distancia mínima entre los dos vértices s y t.*

No hay ningún algoritmo específico para resolver el problema de saber la distancia mínima entre dos vértices de un grafo. Lo que sí que hay es un algoritmo para saber todas las distancias mínimas entre un vértice dado y cualquier otro. La razón es simple. Como para saber cual es el camino más corto entre dos vértices cualesquiera hay que tener en cuenta todo el grafo, entonces ya puestos, resolvemos el problema genérico de una tacada.

5.5.1 Algoritmo de Dijkstra

Edsger W. Dijkstra

Edsger Wybe Dijkstra, con "i", "j", y "k" igual que en el abecedario, (1930-2002) fue un físico teórico holandés. De família intelectual, madre matemática y padre químico, ya de chico era una especie de niño prodigio que estudiaba en escuelas para niños brillantes. Habiendo deseado ser abogado, una vez acabados los estudios de físico teórico se dejó embrujar por la programación computacional. El año 1956 propuso el algoritmo de caminos mínimos que se muestra en esta sección y que recibió el Premio Turing, el de más prestigio en la disciplina algorítmica. Además, también fue Dijkstra quien estableció la inconveniencia de la sentencia GOTO en los programas informáticos, que tantos desastres había provocado. Eso introdujo el uso sistemático de bucles estructurados en los códigos de los programas, y por tanto tuvo un impacto todavía mayor que su algoritmo. Por otra parte se dedicó a la verificación formal de algoritmos, tarea soporífera por antonomásia. La verificación formal pretende demostrar la validez de un algoritmo. Las tentativas que se han ido realizando en este sentido no parece que hayan tenido demasiado éxito, ya que la capacidad expresiva del lenguaje es muy limitada para un objetivo tan ambicioso. Es una buena idea, poder demostrar que un algoritmo funciona. Pero es demasiado difícil. Dijkstra proponía un método que arrancaba en el modelo matemático del problema, y lo sometía a una serie de

transformaciones para hacerlo programable. Muy complicado. Pasó la última etapa de su vida profesional intentando hacer más sencilla la fluidez del lenguaje formal. Pero se debió cansar, ya que se jubiló.

Como se ha dicho más arriba, dado un grafo con pesos, o distancias, y un nodo inicial, el algoritmo de Dijkstra sirve para saber la distancia del camino de mínima distancia, o camino mínimo, entre el nodo inicial y cualquier otro del grafo. Añadiendo las estructuras de datos pertinentes, podemos obtener el conjunto de las aristas de estos caminos.

Para el caso de los grafos dirigidos, ninguna novedad. El mismo algoritmo nos daría las distancias de los caminos mínimos dirigidos, desde el nodo inicial hasta cualquier otro nodo asequible. Si nos insteresase en sentido contrario, o sea, los caminos desde cualquier nodo hasta un nodo final, entonces deberíamos cambiar las direcciones del grafo.

En el Algoritmo 5.3 se puede observar una implementación del algoritmo de Dijkstra. En la cabecera hay dos parámetros de entrada, el grafo g, y el vértice inicial s, y dos más de salida, el vector de distancias d, y el árbol de predecesores p, implementado como siempre en un vector.

De entrada se asocia una distancia a cada vértice del grafo, y se consideran estas distancias como prioridades en una cola de prioridades por mínimos. Los ítems de la cola, ⟨índice,prioridad⟩, representan parejas ⟨vértice,distancia⟩. Adicionalmente, se mantiene la misma información en el vector de distancias, parámetro de salida, que también será el vector solución del problema. O sea, que se guarda las distancias en dos lugares distintos. Por una parte ordenadas según el mismo valor de la distancia, y por otra, según el índice del vértice. Eso hace que cuando se actualiza un ítem en la cola, también se hace en el vector de distancias. La cola se inicializa $\langle i, \infty \rangle$ para todos los vértices del grafo excepto para el vértice inicial, $i \in V, i \neq s$. En consonancia con esto, el vector se inicializa a ∞ para todos los vértices menos el inicial, que se inicializa a 0. Entonces se entra en un bucle en el que a partir del vértice de distancia mínima se cuestiona, en la sentencia alternativa, la posibilidad de reducir las distancias de sus vecinos accediendo a ellos a través de él mismo. En caso afirmativo, actualiza las distancias de estos vecinos, reduciéndolas. Y en consonancia, actualiza las nuevas prioridades en la cola. Eso se hace así hasta que la cola quede vacía.

El papel del vector de predecesores, no es un papel funcional. No condiciona en nada la ejecución, y podría no aparecer si no interesase. Esta estructura es una de aquéllas que tan solo añadimos para extraer alguna información adicional cuando pueda interesar. Del papel superfluo que hacen estas estructuras se ha hablado en detalle en la Sección 5.1.

```
void dijkstra(grafo_pesos& g, int s, int d[], int p[])
{
    int n = g.tamano();
    cola_de_prioridad_minheap Q(n);

    int v;
    para_todo_vertice(v,g) if (v!=s) {
        d[v] = oo;
        Q.insertar(v,oo);
        p[v] = -1;
    }
    d[s] = 0; p[s] = -1;
    Q.insertar(s,0);
    while (!Q.vacia()) {
        item i = Q.getmin();
        int u = i.indice;
        para_todo_vecino(l,g[u]) {
            int v = l→v;
            int w = l→w;
            if (d[v] > d[u] + w) {
                d[v] = d[u] + w;
                Q.actualiza(v,d[v]);
                p[v] = u;
            }
        }
    }
}
```

Algoritmo 5.3 *Caminos mínimos de Dijkstra.*

Se observa también que no se trata la posibilidad de soluciones no factibles. Para el caso del problema de caminos mínimos, una solución no factible sería cuando el grafo fuera disconexo, es decir, que tuviera dos o más componentes conexas. En ese caso, si en un grafo disconexo pretendemos saber la distancia entre dos vértices de componentes distintas, entonces el algoritmo de Dijkstra nos retornaria el valor ∞. Es su manera de expresar la no factibilidad.

En otras implementaciones del Algoritmo 5.3 se pone la sentencia alternativa del núcleo en un procedimiento aparte. Esto se hace así para aislar una operación a la que se acostumbra a denominar *relajación*. Relajar una distancia es actualizarla reduciéndola cuando se ha descubierto un camino de distancia inferior. Volveremos a ello en la Sección 5.6.2.

Para efectuar un análisis de eficiencia del Algoritmo 5.3, tomamos de referencia la parte de la relajación. Contamos cuántas veces se ejecuta la parte interior del *if* en el peor de los casos. Para esto, otra vez utilizamos el teorema de Euler para asegurar que la suma de todos los grados es $\Theta(E)$. Y teniendo en

cuenta que la llamada a la actualización de la cola puede tardar como cualquier inserción, $\Theta(log\ V)$, entre todas las relajaciones que se realizan nos queda una eficiencia perteneciente a $O(E\ log\ V)$.

En la Figura 5.2 se puede contemplar en detalle la evolución de las estructuras de datos a lo largo de una ejecución para un grafo chiquito. Se muestra el estado de las tres estructuras de datos en las primeras cuatro iteraciones. En la Figura 5.2*(1)* hay los valores de los dos vectores y de la cola antes de entrar en el *while*. Estos mismos valores son los que hay después de la primera línea del interior del bucle, en la que extraemos el ítem de más baja prioridad de la cola como vértice actual, u en el Algoritmo 5.3. A medida que se traten los vértices aparecen en trazo grueso en la Figura 5.2. Está bien claro que con la primera extracción de la cola obtendremos el ítem ⟨s,0⟩.

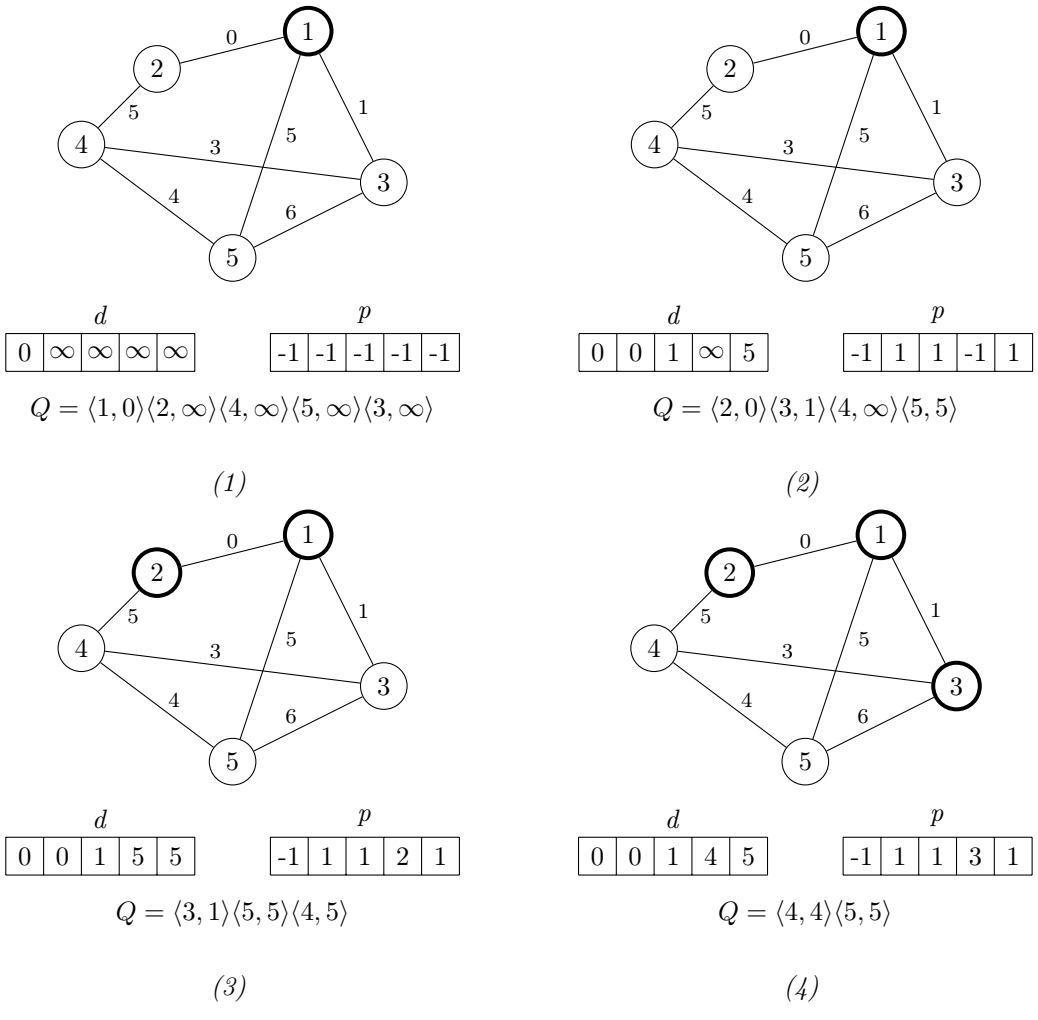


Figura 5.2: *Evolución de las estructuras en el Algoritmo 5.3.*

Procedemos recorriendo los vecinos de este primer nodo $u = s$. En concreto

en el primer caso, la sentencia alternativa será cierta siempre. Todos los vecinos del nodo inicial tienen una distancia infinita, y por tanto, siempre será mayor que cero más el peso de la arista entre el nodo actual y el vecino que estamos tratando. Así pues, una vez actualizadas estas distancias, Figura 5.2*(2)*, salimos del bucle interior y reiteramos el bucle principal. El nuevo vértice actual es precisamente el más próximo al anterior, que en la Figura 5.2*(3)* se ve que es el vértice 2, a distancia 0 del inicial. A partir de este nuevo vértice, visitando todos sus vecinos, damos la bienvenida al vértice 4, aunque sea con distancia 5. El vértice 2 no tiene más vecinos a los que reducir la distancia, ya que el vecino 1 que tiene via arista $\{2, 1\}$ que es su predecesor, ha establecido la distancia a 0 entre los dos nodos, y por tanto, ahora no es inferior. Es más, si el *if* tuviera un "$\geq$" en lugar de un "$>$", el programa, en este punto, se colgaría. En la Figura 5.2*(4)* el vértice actual es el 3, que está más cerca del 1 que el 4 del 2. Con este nuevo vértice actual nos encontramos en un caso que todavía no había ocurrido. La distancia del nodo 4 puede ser reducida de 5 a 4. Es el único caso de este ejemplo que la detección de un ciclo sirve para relajar una distancia. Finalmente, también en la Figura 5.2*(4)*, ya se han encontrado todos los caminos mínimos, y la sentencia alternativa ya no volverá a ser cierta. Por esta razón no se ilustran todas las iteraciones hasta al final. Porque a partir de la situación mostrada en la Figura 5.2*(4)* las cosas ya no cambian más.

Observando el algoritmo de Dijkstra para los caminos de mínima distancia, es fácil pensar que no debería hacer falta tanto trabajo. Parece que si modificásemos ligeramente los algoritmos de recorrido de grafos podríamos extraer la información que nos da este nuevo algoritmo. Parece que si en lugar de considerar que cada arista es una unidad de distancia más lejana de la raíz considerásemos que lo es tantas unidades como la distancia añadida indique, debería funcionar. Pero no es cierto. El comportamiento de los dos algoritmos es totalmente contrario cuando se encuentran con un ciclo en el grafo. Los recorridos lo ignoran, y no siguen por aquel camino. El algoritmo de Dijkstra, en cambio, lo analiza preguntándose si tal vez merece más la pena el nuevo camino encontrado hasta el nodo ya conocido que el camino que se tenía hasta ahora. Con todo, los recorridos encuentran caminos mínimos en número de aristas, que no es lo mismo que en distancias.

Pesos Negativos

A veces hace falta calcular caminos mínimos como subproblemas de otros problemas mayores. Y aparecen grafos con costes negativos en los que hay que calcular caminos de coste mínimo. No es una elucubración matemática. Ocurre.

Una cuestión que parece clara, es que si dispusiésemos, por las razones que fuere, de costes negativos, deberíamos poder sumar una constante igual al valor del coste mínimo, a todos los costes del grafo, solucionar el problema con el Algoritmo 5.3, y después restarle la constante, tantas veces como aristas tenga el camino mínimo encontrado, al valor de la función objetivo. Nada más lejos de la realidad. Como se ve, estamos entrando en un terreno resbaladizo.

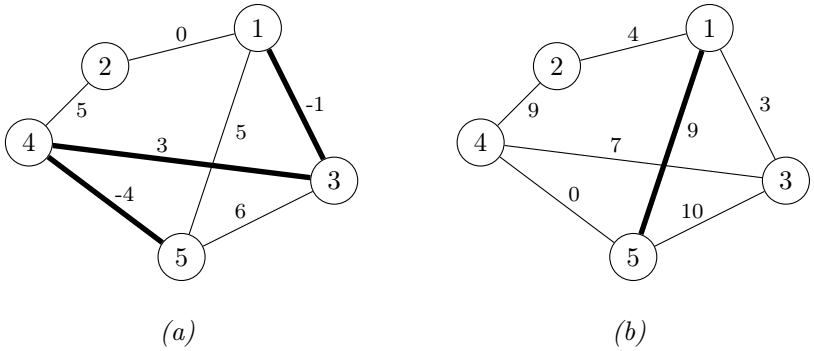


Figura 5.3: *Caminos mínimos en grafos con pesos negativos.*

En la Figura 5.3 tenemos esta idea representada en el grafo del ejemplo. Tal como se puede ver, el grafo de la Figura 5.3*(a)* tiene un par de pesos negativos. En la Figura 5.3*(b)* se ha sumado el mínimo en valor absoluto, 4, a todos los pesos, para que todos sean mayores o iguales a cero.

Ahora observad el coste del camino mínimo entre 1 y 5. En la Figura 5.3*(a)* el camino es $1-3-4-5$ con un coste total de -2. En la Figura 5.3*(b)* el camino mínimo entre 1 y 5 está formado directamente por la arista $1-5$, con un coste total de 9. Así pues, ya se ve que ni tan siquiera las aristas que componen el camino mínimo coinciden.

La dificultad fundamental es que el algoritmo de Dijkstra funciona examinando los caminos por orden de distancia creciente. Se trabaja suponiendo que al añadir una arista a un camino, el coste del camino aumenta. Por eso, esta idea no funciona en absoluto. Los nuevos caminos mínimos en el nuevo grafo no tienen relación alguna con los caminos mínimos en el original. En otras palabras, si al Algoritmo 5.3 le proporcionamos un grafo con pesos negativos, los caminos van a buscar las aristas de pesos negativos, y entonces utilizan más aristas de las que utilizarían si no fuese ese el caso. Por descomptado, estamos suponiendo que el grafo no tiene ciclos de pesos negativos, ya que entonces, cualquier camino que pudiera acceder al ciclo tendría una distancia igual a $-\infty$.

En el código que se sumministra con este libro hay implementada esta idea. De manera que si a la rutina de Dijkstra se le pasa un grafo con pesos negativos se puede comprobar que los resultados son del todo impredictibles. En el fondo, está relacionado con un problema mucho más complejo que el que Dijkstra resuelve. El problema de caminos máximos, del cual podría hablarse en otro libro.

Por si sirve de consuelo, en el próximo capítulo veremos un algoritmo más genérico que el de Dijkstra. El algoritmo de Floyd calcula los caminos mínimos en un grafo entre cualquier pareja de nodos, y además, admite pesos negativos. O sea, que es mucho más genérico. Pero claro, eso utilizando el esquema algorítmico de la programación dinámica. Cosa que significa requiriendo mucho

más espacio, y por tanto, útil para grafos más pequeños.

5.6 Árboles de Expansión Mínima

Un árbol de expansión mínima (*minimum spaning tree*, o directamente MST, en inglés) es un grafo que se define en referencia a otro grafo. Es el resultado de hacer un cálculo a partir de un grafo inicial que nos proporciona como solución otro grafo, que es el árbol. En la Sección 4.1.3 se explica que un árbol es un grafo conexo y acíclico. El relativo *de expansión* es para indicar que el árbol contiene todos los nodos del grafo inicial. Y el adjetivo *mínima* es porque se minimiza la suma de los pesos de todas las aristas seleccionadas.

Definición 5.3 Problema del Árbol de Expansión Mínima. *Siendo $G(V, E)$ un grafo conexo no dirigido, y c una función real de costes no negativa asociada a las aristas, $c : E \to \mathbb{R}^+ \cup \{0\}$, el Problema del Árbol de Expansión Mínima consiste en seleccionar un subconjunto conexo de aristas $T \subseteq E$, de manera que incida en todos los nodos del conjunto V, y la suma de los costes de todas ellas sea mínimo.*

Lo denominamos T de *tree* en inglés.

De utilidades finales tiene muchas. Por ejemplo, en un circuito integrado donde los componentes electrónicos hacen el papel de vértices, necesitamos a menudo que diferentes puntos del circuito estén siempre a la misma tensión eléctrica, debiendo de escoger las conexiones entre las líneas predeterminadas de una malla, ahorrando tanto filamento de oro como sea posible. Asfaltar la primera red de carreteras entre varias poblaciones a partir de los caminos forestales ya existentes, con el mínimo coste posible. O inaugurar cualquier servicio nuevo que se ofrezca para cualquier otro tipo de red, como la televisión digital terrestre. Además, como en el caso de la sección, 5.5.1 y quizás con más razón todavía, el problema de encontrar árboles de expansión mínima prolifera como subproblema en multitud de problemas de diversas disciplinas.

Ya que nos movemos dentro de un espacio condenadamente discreto, de variables enteras, cualquier definición que arrastre aires de continuidad es muy bienvenida. En este sentido, llegados a este punto conviene enunciar un problema conocido como *Árbol de Steiner*. Dado un grafo conexo $G(V, E)$ con aristas de costes no negativos, $c : E \to \mathbb{R}^+ \cup \{0\}$, y un subconjunto de nodos *terminales* $S \subseteq V$, el problema del árbol de Steiner consiste en encontrar el árbol mínimo que una todos los nodos terminales. A los nodos que no son del subconjunto S, se les denomina nodos de Steiner, [9]. Observad pues, que el árbol de Steiner cubre un espacio en medio del de caminos mínimos, cuando $|S| = 2$, y el árbol de expansión mínima, cuando $|S| = n$. Eso significa que si tuviéramos un algoritmo eficiente para resolver el problema del árbol de Steiner, entonces ya no nos haría falta ni el algoritmo de Dijkstra ni los de Kruskal y

Jarník. Desafortunadamente, no tenemos ningún algoritmo así.

En esta sección se ilustran los dos algoritmos más populares para calcular árboles de expansión mínima. Como algoritmos voraces que son, en los dos casos dejan de considerar una arista un vez ha sido considerada. En primer lugar se muestra el que utiliza un método más sencillo, y fácil de recordar. Es el árbol de Kruskal. En este algoritmo las soluciones parciales son definitivas. O sea, una vez se ha decidido que una arista ha de formar parte del árbol, no hay rectificación posible. Ordena las aristas por orden creciente de peso. O sea, postulamos aquí mismo, que las dos aristas de menos coste de un grafo formarán parte de su árbol mínimo. Tres ya no. Podrían formar un ciclo. El inconveniente de Kruskal es que las soluciones parciales no son conexas. Kruskal obtiene un árbol totalmente desarraigado. En cambio, el algoritmo de Jarník actúa topológicament, por medio de las conexiones existentes en el grafo. La construcción del árbol es local. Toda solución parcial es conexa. Y por esta razón, a Jarník le damos un nodo raíz a partir de donde comenzar a construir el árbol.

5.6.1 Algoritmo de Kruskal


Joseph B. Kruskal

Joseph Bernard Kruskal (1928 - ...) es un matemático y estadístico norteamericano. Profundamente interesado en el razonamiento, es un psicometrista de reconocido prestigio. Medir la inteligencia es una idea extraña. Ha presidido asociaciones científicas como la American Statistical Association, y también otras de orientación más humanitaria en soporte de los derechos civiles de las personas. Kruskal es el autor del algoritmo que se explica en esta sección. Seguramente el de más renombre para hacer la tarea que hace, buscar el árbol de expansión mínima de un grafo. Como estadístico ha propuesto resultados de gran impacto en el análisis de varianzas, así como en técnicas de clasificación y de visualización de datos a fin de hacer aflorar relaciones entre ellas. Otros aspectos de carácter social son análisis léxico-estadísticos, y estudio de palabras de raíces parecidas en distintos lenguajes humanos. En esta línea, participa en un proyecto para hacer una base de datos de palabras provenientes del indoeuropeo, la lengua predecesora del latín y del anglosajón. Por parte de madre, ha resultado también ser un experto en papiroflexia.

Para introducirnos en el algoritmo de Kruskal es muy conveniente dar un repaso previo a la Sección 1.1.4 donde se vieron las clases de equivalencia. Y también a su implementación en estructuras de datos, que hemos llamado particiones, en la Sección 2.4. En síntesis, con una esctructura como la *particion* podemos etiquetar un conjunto de elementos. Dispone de dos operaciones características. Podemos ordenar que dos elementos tengan la misma etiqueta con la operación *unir(x,y)* y también podemos preguntar por la etiqueta de cualquier elemento con la operación *representante(x)*. La gracia está en que dos elementos

unidos comparten etiqueta.

Aprovisionados con la estructura apropiada, echemos un vistazo al algoritmo de Kruskal para calcular el árbol de expansión mínima de un grafo conexo. Seguramente este es el algoritmo más famoso para esto, aunque el hecho de no mantener la conectividad entre las diferentes partes del árbol mientras se va construyendo es un inconveniente que lo hace inútil para algunas aplicaciones. Es un procedimiento muy fácil de entender, y más fácil todavía de recordar.

Kruskal comienza ordenando las aristas por orden creciente de coste. Inicializa la partición del conjunto de nodos, de manera que cada nodo es de su propia clase. Entonces va tomando las aristas de una en una. Si conectan vértices de distintas clases, pasan a formar parte del árbol solución, y se unen sus dos nodos en la partición. Y si no, o sea si pertenecen a la misma clase, entonces nada, se descarta la arista.

En el Algoritmo 5.4 hay una implementación de la idea de Kruskal. La cabecera tiene como parámetro de entrada el grafo con pesos g, y de salida, el grafo con pesos que será el árbol de expansión mínima, t.

```
void kruskal(grafo_pesos& g, grafo_pesos& t)
{
    int n = g.tamano();
    cola_de_prioridad_minheap Q(n*n);

    int u;
    para_todo_vertice(u,g) {
        para_todo_vecino(l,g[u]) {
            int v = l→v;
            if (u<v) Q.insertar(u * (n+1) + v,l→w);
        }
    }

    particion p(n);
    while (!Q.vacia()) {
        item i = Q.getmin();
        int u = i.indice / (n+1);
        int v = i.indice % (n+1);
        if (p[u] != p[v]) {
            t.anadir_arista(u,v,i.prioridad);
            p.unir(u,v);
        }
    }
}
```

Algoritmo 5.4 *Árbol de expansión mínima de Kruskal.*

El Algoritmo 5.4 usa una cola de prioridad de mínimos para ordenar las aristas. La inicializa con una cota para el número de aristas, $n * n$. En una implementación comercial podría hacerse con un quicksort si el algoritmo se tuviera que ejecutar de forma masiva. Aquí se utiliza una cola de prioridad para no añadir más código de este capítulo en el que se subministra con este libro. Los programas sumministrados son independientes por capítulos, y no utilizan ninguna librería. Por esta razón no se ha querido implementar más de un código de ordenación para los algoritmos voraces. Además, como no tenemos parametrizado el tipo de los elementos de la cola, y están fijados a ítems con un entero y un valor real, nos vemos forzados a hacer alguna filigrana con la codificación de los dos vértices de cada arista en la cola. Eso es, codificamos el índice correspondiente a la arista (u, v) con el entero $u(n+1)+v$, siendo n el número de nodos. Hay que tener en cuenta que sólo hay que añadir una vez las aristas a la cola. Por eso sólo se añaden cuando el primer vértice es menor que el segundo. Esto significa que el Algoritmo 5.4 sirve para grafos no dirigidos exclusivamente. De hecho, el concepto de árbol de expansión mínima también va ligado a grafos no dirigidos.

Con todas las aristas en la cola, preparadas para ser solicitadas por orden ascendiente de peso, declaramos la partición con tantos elementos como nodos en el grafo. Guardaremos las clases de equivalencia de la relación "están conectados en la misma componente conexa del árbol solución". O sea, en todo momento, dos vértices tendrán el mismo representante si ya están conectados por algún camino. Recordad que los claudátors para las particiones son un operador para embellecer la función *representante*.

Con todo, entramos en el bucle principal del esquema voraz. A partir de aquí, tan solo tomando la siguiente arista de mínimo peso, y aceptándola cuando una nodos de diferentes clases o rechzándola si los nodos que une ya son de la misma clase, construiremos el árbol. Bien, eso siempre que el grafo de entrada sea conexo. Si no es el caso, entonces saldremos del Algoritmo 5.4 con un bosque. Un árbol para cada componente. En el Algoritmo 5.4, la implementación del árbol resultante se ha hecho en una estructura de grafo, y no en un vector de predecesores como es habitual. Así podemos guardar de una manera elegante los pesos de las aristas seleccionadas. Se hubiera podido hacer en un vector adicional de valores reales, que en cada posición guardaría el peso de la arista de cada nodo con su predecesor en el árbol. Entonces el algoritmo hubiera resultado más eficiente, aunque no tan sencillo de interpretar.

O sea, que el algoritmo de Kruskal calcula el árbol en base a seleccionar aristas por orden de peso. Esto provoca que en los estadios intermedios tengamos un bosque. Un conjunto de árboles disjuntos que iteración a iteración, se van uniendo.

En la Figura 5.4 se puede hacer el seguimiento del algoritmo en las iteraciones más relevantes. La Figura 5.4*(1)* muestra el estado inicial antes de entrar en el bucle principal. No aparece el contenido de la cola de prioridades porque es demasiado larga. En cambio, sí que aparece un vector con los representantes de los nodos que viene implementado en la partición. Inicialmente cada nodo es el representante de su propia clase.

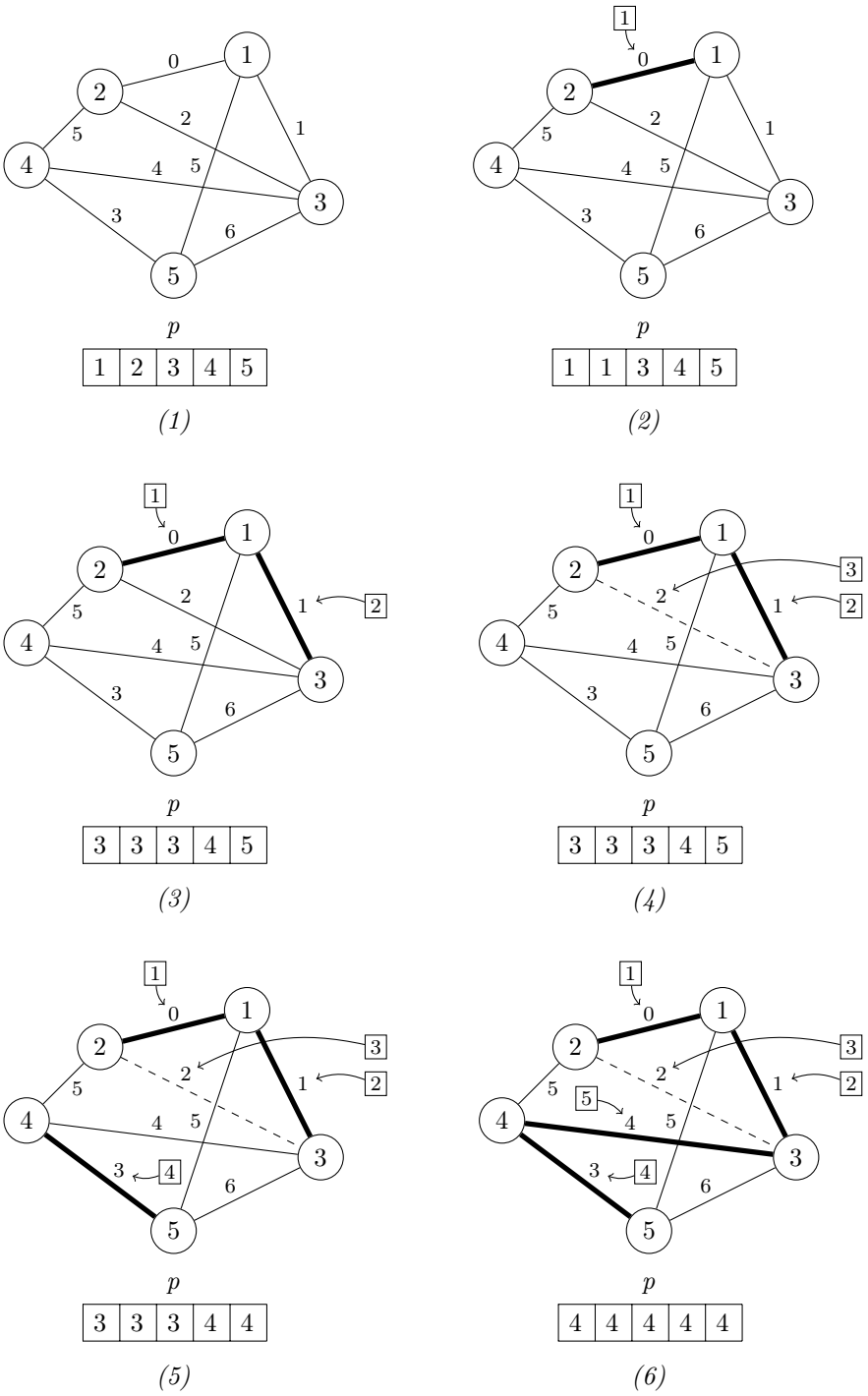


Figura 5.4: *Árbol de expansión mínima de Kruskal*

Entonces, se entra en el bucle con las aristas ordenadas y con la partición así.

En las siguientes partes de la Figura 5.4 se indica en un cuadradito y una flecha cuál es la arista que se ha analizado en cada iteración. Las aristas en negrita, a lo largo de toda la figura, son las seleccionadas para formar el árbol resultante. Las rechazadas aparecen en trazos discontinuos. Se puede observar que el orden de selección de las aristas a tratar se corresponde con el orden creciente de los pesos. Por otra parte, también se puede deducir de la evolución del contenido de la partición que cada vez que une dos vértices se ha optado por poner de representante al de la clase menos poblada, aquél que tenga menos elementos en su clase.

En la Figura 5.4*(2)* ya se ha tratado la primera arista. Por eso aparece el cuadradito con un 1 señalando la arista de peso menor del grafo, que se ha seleccionado como miembro del árbol resultante, y se han unido los dos vértices que contiene. En este momento tenemos cuatro clases de equivalencia diferentes, una que contiene los nodos 1 y 2, y las otras tres que son una de cada vértice, 3, 4, y 5. Por otra parte, ya tenemos también una arista, la $\{1, 2\}$, que definitivamente aparecerá en la solución.

La siguiente iteración se muestra en la Figura 5.4*(3)*. Se ha seleccionado la arista de menos peso de las restantes. Es la arista $\{1, 3\}$, con peso 1. Por eso aparece el cuadradito con el 2 indicando que es la segunda acción que se realiza, y la arista en negrita indicando que a partir de ese momento forma parte también de la solución. Debajo, la partición ha unido dos de los cuatro grupos que tenía antes. Quedan tres.

Procedemos hacia la tercera iteración. En la Figura 5.4*(4)* se produce un fenómeno nuevo. De entrada tomamos la arista de menos peso de las que quedan, que es la $\{2, 3\}$, con peso 2. Entonces la sentencia alternativa del interior del bucle del Algoritmo 5.4 falla. Retorna falso. Resulta que los dos vértices forman parte de la misma clase de equivalencia. Si añadiésemos esta arista a la solución formaríamos un ciclo. Y un árbol no tiene ciclos. Total, la descartamos. Definitivamente, la arista $\{2, 3\}$ no formará parte del árbol solución. Por eso aparece en trazos discontinuos. Igualmente pero, tiene el cuadradito indicando que rechazarla ha sido la tercera acción que hemos efectuado. Esta vez el contenido de la estructura partición no ha variado.

Bien, continuamos con la próxima iteración, y otra vez aparece un fenómeno que no había sucedido previamente. La arista seleccionada esta vez es la arista $\{4, 5\}$. Como se ve, es la cuarta acción que se hace. La ponemos en negrita porque entra a formar parte de la solución. En este momento las aristas del árbol forman un bosque disjunto. Así es como trabaja Kruskal. Como se puede ver también en la Figura 5.4*(5)*, las dos clases de los nodos 4 y 5 se han unido. Ya tan solo quedan dos clases de equivalencia. Lógicamente, el número de clases existentes en la partición se corresponde con el número de árboles en el bosque, o de componentes conexas en el grafo solución.

Y ya, en la Figura 5.4*(6)*, la arista de menos peso de las tres que quedan se añade a la solución. La $\{3, 4\}$. Hace que se unan en un solo árbol las dos componentes que se habían creado en las iteraciones anteriores. También hace que todos los vértices pertenezcan a la misma clase de equivalencia.

Llegados a este punto, la sentencia alternativa no volverá a ser cierta. Las cosas quedarán igual hasta al final, excepto que antes, todavía se descartarán las dos aristas que en la figura no se han tratado.

Podemos analizar la eficiencia del algoritmo de Kruskal teniendo en cuenta que la ordenación de las aristas supone $\Theta(E \log E)$. De hecho, las implementaciones de las operaciones de la partición son logarítmicas con el número de nodos, $\Theta(\log V)$. Por tanto, no cargan la complejidad del algoritmo, que viene dominada por la ordenación inicial.

5.6.2 Algoritmo de Jarník, o Prim

Vojtech Jarník

Vojtech Jarník (1897-1970) fue un matemático checo. Adentrado en la teoría de los números y el análisis matemático, dedicó esfuerzos al problema del círculo Gaussiano. Eso es, a la relación entre el número de puntos con coordenadas enteras y el círculo que los contiene. Se dice de Jarník que era un hombre muy divertido. Tenía un gran sentido del humor. Incluso, hizo una demostración por inducción matemática de los problemas que tendría Vladimir Lenin con sus seguidores. Fue maestro durante cuarenta y siete años de la Charles University de Praga. Según [20], Jarník era un profesor amable capaz de transmitir su entusiasmo por las matemáticas a los estudiantes. Su enorme erudición, su tacto, y el carácter humano provocaban admiración y un respeto profundo a cualquiera que le hubiese conocido personalmente.

Probablemente, Jarník no era consciente de la trascendencia de su algoritmo para encontrar el árbol de expansión mínima de un grafo. Fue publicado el año 1930. De todo su legado, parece que una cosa sin importancia fuera este algoritmo. Unos cuantos años más tarde, en 1956, la misma idea fue redescubierta por Robert Clay Prim, un compañero de Kruskal, matemático e ingeniero de computación norteamericano.

La mejor manera de comprender el algoritmo de Jarník es a partir del de Dijkstra. De hecho, trabaja exactamente igual. La única diferencia es que con el algoritmo de Dijkstra nos guardamos las distancias de cada nodo al nodo inicial. En el de Jarník, en cambio, en lugar de al nodo inicial, nos guardamos la distancia de cada nodo al nodo que lo une al árbol, es decir, al predecesor. Esto provoca que para el caso del algoritmo de Jarník se tengan que controlar

necesariamente los vértices tratados, cosa que no ocurría en el algoritmo de Dijkstra porque todas las distancias se referían al vértice inicial.

Hay una diferencia importante entre el algoritmo de Jarník, y el de Kruskal. Así como en el de Kruskal las aristas eran añadidas o descartadas de la solución, y no había más vuelta de hoja (comportamiento voraz por antonomasia), para el caso de Jarník una arista pertenece a la solución mientras no aparezca una mejor. O sea, que se podría enmarcar legítimamente en la programación dinámica. En ese caso, como en el algoritmo de Von Newmann, este ejemplar habría existido antes de la formalización de su esquema algorítmico. Sin embargo, tradicionalmente, se ha considerado como un algoritmo voraz, aunque como se acaba de mencionar, podría ser considerado perfectamente un algoritmo de programación dinámica.

En el Algoritmo 5.5 se puede ver una implementación sencilla del algoritmo de Jarník.

```
void jarnik(grafo_pesos& g, int s, int p[], double d[])
{
    int n = g.tamano();
    cola_de_prioridad_minheap Q(n);

    int u;
    para_todo_vertice(u,g) if (u != s) {
        d[u] = oo;
        Q.insertar(u,oo);
    }
    Q.insertar(s,0);
    p[s] = NULL;
    d[s] = 0;
    while (!Q.vacia()) {
        item i = Q.getmin();
        int u = i.indice;
        para_todo_vecino(l,g[u]) {
            int v = l→v;
            double w = l→w;
            if (Q.hay(v) && w < d[v]) {
                p[v] = u;
                d[v] = w;
                Q.actualiza(v,d[v]);
            }
        }
    }
}
```

Algoritmo 5.5 *Árbol de expansión mínima de Jarník.*

Es casi clavado al Algoritmo 5.3, el de Dijkstra, con tan solo dos diferencias. Ambas en la misma línea de la condición de la sentencia alternativa. Estas dos diferencias se comentan seguidamente.

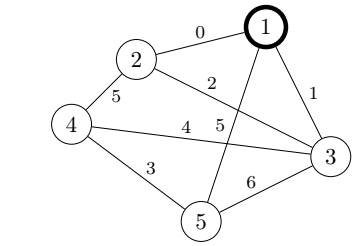
Por una parte, el algoritmo de Jarník controla que los vértices no hayan sido tratados previamente. Eso con Dijkstra no era preciso. En particular, en esta implementación, se efectúa consultando si ya han desaparecido de la cola, cosa que tarda un tiempo $\Theta(n)$ fácilmente mejorable con un vector de booleanos, que lo convertiría en $\Theta(1)$. Una implementación más eficiente del Algoritmo 5.5, pues, utilizaría un vector de booleanos. Este vector se inicializaría a falso para todos los nodos excepto para el nodo raíz, y se establecería a cierto, para cada nodo, al salir del bucle más interno. Entonces la condición del *if*, en lugar de decir $Q.hay(v)$, se debería de preguntar si el nodo v es no visto. Esta nueva versión sería más eficiente, sin duda. De todas maneras, se ha presentado así porque parece más fácil de entender, sin el ruido provocado por variables prescindibles. No obstante, debe quedar claro que en el Algoritmo 5.5 se ha sacrificado eficiencia para ganar legibilidad.

Y por otra parte, la segunda diferencia importante respeto al algoritmo de Dijkstra es el cambio de significado del vector d de distancias. En Dijkstra, significaba la distancia de cada nodo al nodo inicial. Ahora significa la mínima distancia de cada nodo al árbol. Es decir, mínima entre todos los nodos del árbol. Por este motivo aquí sí que es preciso saber los nodos tratados, porque si no, en lugar de un árbol nos resultaría un conjunto disconexo de nodos aparejados o en grupos de tres, con las aristas de mínimo coste. A diferencia del Algoritmo 5.4 de la Sección 5.6.1, donde el árbol resultante era implementado en un grafo, en el Algoritmo 5.5 se implementa en un vector de predecesores.

En la Figura 5.5 se puede hacer un seguimiento del código del Algoritmo 5.5. Para cada iteración, las tres estructuras de datos que se utilizan aparecen bajo el grafo. El vector de distancias al árbol, el de predecesores, y la cola.

En la Figura 5.5*(1)* se muestra el estado después de la operación extraer el mínimo de la cola de prioridad, que ha sido el ítem $\langle 1, 0\rangle$. Tanto el vector de distancias como el de predecesores mantienen todavía los valores de la inicialización. La cola, en cambio, ya ha variado porque se ha extraído ese primer elemento.

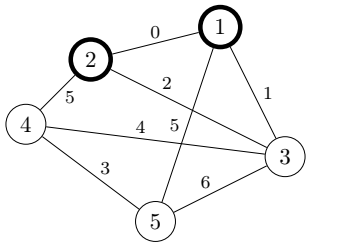
En la iteración siguiente, Figura 5.5*(2)*, se han actualizado los dos vectores según los pesos del corte del nodo 1. Se puede observar el estado después de haber extraído el elemento $\langle 2, 0\rangle$ de la cola. Eso ha provocado que el nodo actual sea el nodo 2.



	d					p				
	0	∞	∞	∞	∞	0	-1	-1	-1	-1

$Q = \langle 2, \infty\rangle\langle 4, \infty\rangle\langle 5, \infty\rangle\langle 3, \infty\rangle$

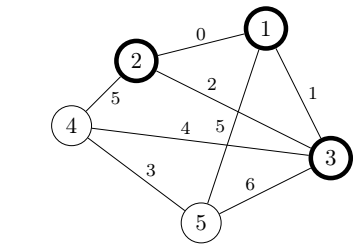
(1)



	d					p				
	0	0	1	∞	5	0	1	1	-1	1

$Q = \langle 3, 1\rangle\langle 5, 5\rangle\langle 4, \infty\rangle$

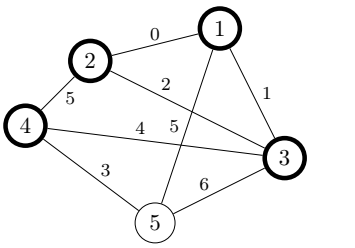
(2)



	d					p				
	0	0	1	5	5	0	1	1	2	1

$Q = \langle 5, 5\rangle\langle 5, 5\rangle$

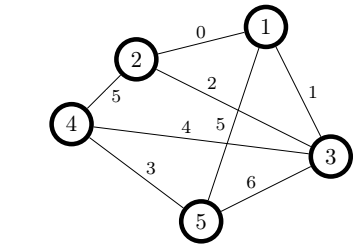
(3)



	d					p				
	0	0	1	4	5	0	1	1	3	1

$Q = \langle 5, 5\rangle$

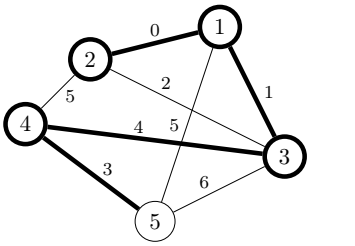
(4)



	d					p				
	0	0	1	4	5	0	1	1	3	4

$Q = \emptyset$

(5)



	d					p				
	0	0	1	4	3	0	1	1	3	1

(6)

Figura 5.5: *Árbol de expansión mínima de Jarník.*

Es por esa razón que en la Figura 5.5*(3)* ha sido actualizada la distancia del vértice 4 al grafo. Porque es vecino del 2. También en la Figura 5.5*(3)*, se ha seleccionado el nuevo vértice actual, 3, quitándolo de la cola. Eso ha provocado que el ítem $\langle 4, 5\rangle$ acabado de actualizar haya pasado adelante. Una consecuencia de tratar el nodo 3 es que la distancia del nodo 4 se reduce de 5 a 4. O sea, hasta ahora, la arista que unía el nodo 4 al árbol era la $\{4, 2\}$. A partir de ahora, y mientras no aparezca una mejor, será la $\{4, 3\}$.

Este efecto se puede ver en la Figura 5.5*(4)*, en la que también se ha actualizado el predecesor del nodo 4, y se ha extraído de la cola el mismo nodo 4 para hacer de nodo actual.

En la Figura 5.5*(5)*, se ha ejecutado de nuevo la operación de relajación para el nodo 5. O sea, se le ha reducido la distancia al árbol gracias a la aparición de una nueva arista candidata. En otras palabras, hasta ahora el vértice 5 estaba conectado vía arista $\{5, 1\}$ de peso 5. Ahora este peso ha podido ser reducido a 3 vía arista $\{5, 4\}$. Finalmente, se quita el ítem del nodo 5 de la cola. No tiene ningún vecino que esté en Q, y el procedimiento acaba.

La eficiencia del algoritmo de Jarník es exactamente la del de Dijkstra, $\Theta(E \log V)$.

En este capítulo se ha visto la aproximación más grosera para atacar problemas de optimización, los algoritmos voraces. Se ha introducido también el principio de optimalidad que se ha utiilizado para demostrar la calidad de la solución de un problema paradigmático de este esquema algorítmico, las gasolineras. También se ha analizado la eficiencia de algoritmos clásicamente enmarcados dentro de la estrategia voraz.

En definitiva, no se puede pasar página si no se mantiene en la cabeza que los algoritmos voraces sirven para dar soluciones de calidad cuestionable en tiempos razonables.

Capítulo 6

Programación Dinámica

Figura 6.1: *Calendario perpetuo.*

Podríamos simplificar la medida del tiempo. Podríamos tener días de diez horas si cada hora midiera 2.4 horas de las de ahora. Horas de diez minutos, minutos de diez segundos... en fin, un reloj no es más que un contador raso y llano. La única diferencia entre contar los números naturales y contar el tiempo està en la base de numeración. El número que representa la hora que es, es un número de seis cifras. La primera cifra son las decenas de hora, que se cuentan en base tres. La segunda, las unidades de hora, depende de la primera, de las decenas. Si las decenas de hora valen menos de

dos, entonces las unidades de hora cuentan en base diez, pero si las decenas de hora son dos, entonces las unidades de hora cuentan en base cuatro. Después vienen las decenas de minuto, en base seis. La siguiente cifra son las unidades de minuto que cuenta en base diez. La penúltima, las decenas de segundos, cuenta otra vez en base seis. Y la última, las unidades de segundo, también cuenta en base diez. Aparte de que los días de la semana cuentan en base siete, tenemos meses de veintiocho, veintinueve, treinta y treinta y un días! Suerte que de meses para arriba, ya todo es en decimal. Años, décadas, siglos, milenios,... Y por abajo igual. Más pequeño que un segundo, ya todo también es en base diez. Décimas, centésimas, milésimas... Los humanos nos complicamos, pero no tanto!

La programación dinámica es especialmente útil para contar en sistemas de numeración donde la base de la cifra dependa de la posición que ocupa. En el sistema de numeración decimal todas las cifras de los números cuentan en base diez. En un reloj ya hemos visto que no. Si nos cuesta saber cómo se representará una cifra en un sistema de numeración de éstos tan complicados, la mejor manera es contando hasta al número. A no ser que tengamos un calendario perpetuo, es difícil saber si el 20 de febrero del año 6470, por decir alguna cosa, será lunes, martes o qué. En cambio, si sabemos que el 19 de febrero del 6470 es miércoles, entonces es un problema fácil. El 20 de febrero será jueves. En esta línea trabaja la programación dinámica.

Eso de que será jueves es cierto. En el calendario perpetuo de la página anterior pone: FIND THE LETTER OF THE YEAR AND THE SAME LETTER OF THE CENTURY. JOIN THE COLUMN WITH THE MONTH. DIVIDE THE CENTURY BY 400 AND LOOK UP THE LETTER IN THE REMAINDER COLUMN. RED MONTH FOR LEAR YEAR. O sea, como dividiendo 6400 entre 400 el resto es cero, utilizamos la columna de más a la derecha de las letras mayúsculas, que si tuviese título sería 000, pero no tiene. Si el siglo módulo 400 fuese 100, deberíamos usar la columna de más a la izquierda... Bien, buscamos el 70 en la tabla central, que por cierto, me gustaría saber como está calculada. Está en la última fila entre el 64 y el 81. Con la columna de la derecha intersecciona con una D. Como el 6470 no será un año bisiesto, utilizamos, del arco superior donde hay los nombres de los meses, los que están en color oscuro, en negro. Hacemos girar el contenido de las dos áreas en forma de arco con el disco de la parte posterior poniendo el febrero bajo la D, y nos queda el número del día 19 bajo la letra W de miércoles en la parte arqueada inferior. No sé si esta tabla se podría calcular con programación dinámica, pero en cualquier caso me parece fascinante. En particular, que el número del medio no sea necesario, y se pueda poner el tornillo que articula los giros de los discos.

Volvamos al tema. Tal como se ha introducido en el capítulo anterior, nos disponemos a tratar problemas que son secuencias de decisiones. En algunos, las decisiones son independientes las unas de las otras. Entonces podemos utilizar el esquema voraz, que se explica en el Capítulo 5. Sin embargo, a veces están íntimamente relacionadas y la única manera de resolverlos es probando todas las posibilidades. Eso se hace con el esquema de búsqueda exhaustiva, que por esta razón también se llama esquema enumerativo, del Capítulo 7. Y otras veces, podemos hacer depender la secuencia de decisiones tan solo de la mejor solución

obtenida hasta el momento, manteniendo abierta la posibilidad de que cualquier decisión tomada pueda ser rectificada ante la aparición de nuevos candidatos. No es como el caso de los algoritmos de Dijkstra o Jarník, donde, cuando todos los vecinos de un nodo han sido analizados, ya sabemos seguro que por lo que respecta aquel nodo, no habrá más decisiones posibles a tomar. No. Ahora, mantenemos abierta la posibilidad de que cualquier nuevo candidado nos haga cambiar todas las decisiones anteriores, si de eso resulta una mejora en el valor de la función objetivo. Para todo esto, tenemos la programación dinámica.

El capítulo abre con una breve introducción histórica de esta técnica donde se presenta Bellman, el del principio de optimalidad. Se habla de la relación que tiene la programación dinámica con la estrategia de dividir y vencer, y se enmarca en las técnicas de optimización. A continuación se establecen un par de estructuras de datos para los algoritmos posteriores, y también se profundiza en el concepto de recurrencia dado en el Capítulo 1. Enmarcado en las recurrencias, se muestra por fin el algoritmo para resolver los números de Fibonacci, y se extrae todo el jugo posible, generalizando todo aquello que comparte con los otros algoritmos de esta técnica. Entonces se muestra el esquema algorítmico y se hacen algunos comentarios para su interpretación. Se dan referencias fisiológicas del aspecto paradigmático que adquieren los algoritmos desarrollados con esta metodología. El capítulo sigue presentando algunos problemas. Primero, el problema del número de subconjuntos, que también se conoce como los coeficientes binomiales, o números combinatorios. Después, un problema llamado *devolver cambio* servirá para contrastar las reflexiones que se habrán hecho sobre el esquema algorítmico. Se implementa también el código para el problema de la mochila, del que tanto se ha hablado en el Capítulo 5. La diferencia más importante entre esta implementación y la voraz es que aquí sí obtendremos la solución óptima. El capítulo cierra con el algoritmo de Floyd para las distancias mínimas entre cualquier pareja de vértices en un grafo conexo no dirigido.

6.1 Introducción


Richard E. Bellman

Richard Ernest Bellman (1920-1984) fue un matemático norteamericano de los primeros en dedicarse a la matemática aplicada, disciplina en la cual se estudian los problemas de decisión multietapa. El nombre dice mucho, problemas de decisión multietapa. En estos problemas hay un conjunto de decisiones a tomar, y, de la toma de las cuáles, se deriva un resultado numérico que se trata de optimizar. O sea, los problemas de optimización lo son. Bellman también tiene resultados relacionados con las cadenas markovianas, aquéllas en las que un evento tiene una probabilidad condicionada por el evento inmediatamente anterior en una sucesión. Esto está relacionado con procesos estocásticos y matemática de mercado. Bellman se erigió como padre de la programación dinámica el año

1953, y fue uno de los pioneros en el ámbito la investigación operativa.

En la Sección 2.2 de la página 59 se ha hecho un poco de historia de la programación de computadoras. También se ha mencionado el concepto de memoria dinámica. Y la antigüedad de su uso. Podría pensarse que con el término *dinámica* se pretende reflejar que la cantidad de memoria utilizada va creciendo a lo largo de la ejecución. Y no se iría desencaminado, ya que el hecho es éste. Esta metodología es fruto de los recursos que utiliza. Es como si fuesen los ordenadores los que nos hubiesen enseñado a nosotros la manera de resolver algunos problemas. Sin embargo, el adjetivo *dinámica* para esta técnica tiene unas razones mucho más prosaicas. Bellman encuñó el término para poder pedir ayudas para la investigación a un secretario de defensa del presidente Eisenhower, que se dice que era alérgico a la palabra "investigación".

Por un lado, la programación dinámica está emparentada con la técnica de dividir y vencer, ya que las dos resuelven subproblemas pequeños para obtener la solución del problema original. No obstante, difieren en la perspectiva. Dividir y vencer es una estrategia descendiente. Comienza fragmentando la instancia inicial hasta tener subproblemas lo bastante chiquitos. La programación dinámica, en cambio, es una técnica ascendiente. Comienza resolviendo los subproblemas más pequeñines y va haciendo crécer el tamaño del subproblema resuelto, hasta llegar a la instancia original. Es como una bola de nieve. Por otra parte, la programación dinámica está contenida en las técnicas focalizadas en la optimización, o sea, de maximización o minimización de alguna función objetivo.

6.2 Vectores Dinámicos y Matrices Dinámicas

A lo largo de este capítulo se utilizarán las estructuras de datos de los Algoritmos 6.1 y 6.2. Llegados a este punto, resulta del todo aconsejable utilizar plantillas, *templates*. Sin ninguna explicación addicional, puede sobreentenderse su funcionamiento.

En adelante, pues, se utilizarán plantillas para parametrizar los tipos. Los objetos creados con plantilla se declaran con el prefijo *template*, y entre ángulos <...> se expresa el parámetro formal del tipo parametrizado, que en el Algoritmo 6.1 es T. Que trascienda, tan solo hay el hecho de que en las próximas páginas nos encontraremos con vectores declarados como *vector<int>*, *vector<double>*, o lo que haga falta.

En la implementación del vector genérico del Algoritmo 6.1 hay dos variables miembro, la dimensión n, y un apuntador a los valores que forman el contenido, v. El constructor admite la dimensión, y un valor del tipo actual para inicializar todas las componentes con este valor. Para esto, se llama a una función *crea()* que permitirá crearlos fuera del ámbito de la declaración. Igualmente, para la destrucción.

Las otras funciones son embellecedoras, en pro de la legibilidad de los algoritmos. Hay el operador de indexación, con claudátors [ ], que sirve para acceder directamente a las componentes, sin tener que usar la variable v. También hay dos tipos de asignaciones definidas. El primero es para inicializar los vectores de esta estructura con vectores primitivos preexistentes. La segunda para asignar vectores de este mismo tipo. Como se ve, copia todos los valores, son asignaciones de contenido, no de referencia.

```
template<typename T> struct vector {
    int n;
    T* v;
    vector<T>(int _n = 0,T x = NULL) { crea(_n,x); }

    void crea(int _n = 0,T x = NULL) {
        n = _n;
        v = new T[n];
        memset(v,x,n*sizeof(T));
    }
    void destruye() { delete [ ] v; }

    T& operator[ ](int i) { return v[i]; }

    vector& operator=(T* p) {
        memcpy(v,p,n*sizeof(T));
        return *this;
    }
    vector<T>& operator=(vector<T> _v) {
        destruye();
        crea(_v.n,NULL);
        memcpy(v,_v.v,n*sizeof(T));
        return *this;
    }
    void in(int MAX = 0) {
        for (int i=0; i<n; i++)
            if (MAX == 0) cin » v[i];
            else v[i] = rand() % MAX;
    }
};
```

Algoritmo 6.1 *Vector genérico.*

Finalmente, hay una función miembro que permite inicializarlo de dos formas más, *in()*. Esta función recibe un parámetro, MAX, que si es diferente de cero, llena aleatoriamente el vector con valores de 0 a MAX. Si el parámetro es cero, entonces los valores para inicializar el vector se obtienen de teclado con la instrucción *cin*. En el código que se subministra con el libro, se acostumbra a

inicializarlos de la primera manera de las tres. O sea, a partir de vectores primitivos preexistentes, ya que en muchos casos, llenarlos aleatoriamente conduciría a la no factibilitad de los problemas que se resuelven.

Para utilizar la estructura del Algoritmo 6.1 hay que declarar la dimensión del vector en el momento de crearlo. Como ya se había mendionado anteriormente, que en la creación se reserve la memoria y después no se pueda cambiar de tamaño, hace que también se las denomine estructuras semidinámicas.

En el Algoritmo 6.2 se muestra la implementación de una matriz con el tipo parametrizado. Se implementa como un vector de vectores, haciendo uso del código del Algoritmo 6.1. Para las matrices, se definen operadores análogos a los de los vectores, y además, una inicialización en la que el contenido de la matriz completa se pasa en un buffer serializado por filas.

```
#include "vector.h"

template<typename T> struct matriz {
    int m;
    int n;
    vector<T>* v;

    void crea(int _m = 0, int _n = 0, T x = NULL) {
        m = _m;
        n = _n;
        v = new vector<T>[m];
        for (int i=0; i<m; i++) v[i].crea(n,x);
    }

    void destruye() {
        for (int i=0; i<m; i++) v[i].destruye();
        delete [ ] v;
    }

    vector<T>& operator[ ](int i) { return v[i]; }

    matriz(int _m = 0, _n = 0, T x = NULL) { crea(_m,_n,x); }

    matriz<T>& operator=(T* buffer) {
        for (int i=0; i<m; i++) v[i] = &buffer[i*n];
        return *this;
    }
    void in(int MAX = 0)
        { for (int i=0; i<m; i++) v[i].in(MAX); }
};
```

Algoritmo 6.2 *Matriz genérica.*

Estas implementaciones no serían precisas si en este libro se hiciera un uso más insistente de librerías estándar de más alto nivel. En particular, de la *std*. Se muestran por su simplicidad y por tener un soporte fundamentado en el lenguaje C++ más primario, que es lo que a lo largo de todo el libro se trata de utilizar. Todo tan sencillo como sea posible. De paso, tener nuestra implementación personal nos facilitará la depuración. Trabajar con la std tiene muchas ventajas, aunque la depuración se transforma en un inconveniente notable.

Tal como se ha indicado en el preámbulo, el uso de plantillas se ha pospuesto tanto como ha sido posible. La sintaxis usada es clavada a la de la std, de manera que el código que se presenta con el libro puede ser compilado cambiando tan solo los archivos de include.

6.3 Recurrencias

En la Definición 1.8 de la Sección 1.6 se ha visto que una recurrencia es una ecuación que relaciona una función real de variables enteras para un cierto valor de las variables, con la misma función para otros valores más pequeños de las mismas variables. En el Capítulo 1, también se ha hablado de recurrencias substratoras y divisoras cuando se veían los Teoremas Maestros.

En este capítulo se trata con recurrencias substractoras. No es que la programación dinámica se dedique exclusivamente a este tipo de recurrencias. Sencillamente, es que son más habituales. Lo cierto es que toda la teoría que se expone podría implementarse para problemas con recurrencias divisoras.

Partimos de una función $f : \mathbb{N}^r \to \mathbb{R}$. Para el caso que $r = 2$, f es una recurrencia si se puede expresar de la forma

$$f(n,k) = g(n,k,f(n-i,k-j)), \tag{6.1}$$

para alguna función g y cualquier $0 \leq i \leq n$ y $0 \leq j \leq k$.

La programación dinámica se dedica a resolver problemas que se pueden plantear como una recurrencia. Viendo la expresión (6.1) cualquiera diría que la implementación de este tipo de algoritmos es recursiva. Y se equivocaría.

En esta técnica, neutralizamos la recursividad natural del problema guardándonos los resultados óptimos anteriores, que normalmente son inferiores y crecientes, en una tabla de dimensión r. O sea, si r es 1, en un vector. Si r es 2, en una matriz, y así. Ya se ve que esto no puede crecer demasiado. En ninguno de los problemas que se acostumbran a analizar cuando se estudia esta metodología, $r > 3$. Eso significa que la estructura de datos más engorrosa que

se utilizará será una matriz de tres dimensiones. Pero de hecho, el esquema es genérico.

Los valores con los que se rellenan las tablas son valores de la función objetivo. Las reglas que guían cómo rellenar la tabla son recurrencias. Eso hace que al final de la resolución haya la recursividad implícita en el contenido de estas tablas. Conviene pensar que el hecho de que sea necesariamente una estructura de datos dinámica, que su tamaño depende del de la instancia del problema que resuelve, le da nombre al esquema algorítmico. Con todo, esta técnica provocará costes importantes de espacio. La programación dinámica es una técnica costosa tanto en espacio como en tiempo. Aún así, se consiguen mantener ambas eficiencias bajo márgenes polinómicos.

Seguidamente se presenta el ejemplo de Fibonacci que es el algoritmo más sencillo que utiliza este esquema algorítmico. Su sencillez nace del hecho de que $r = 1$, de manera que la estructura utilizada es un vector. De hecho, como se verá cuando se aborde el tema de las mejoras sistemáticas, ni siquiera el vector es preciso.

6.3.1 Fibonacci

Está bien claro que el algoritmo para calcular el enésimo término de Fibonacci, ya visto en las Secciones 1.6.3 y 3.1.3, se ajusta perfectamente a la expresión (6.1). Tal como se ha dicho en el Capítulo 3 la técnica más adecuada para resolver el problema de Fibonacci es la programación dinámica, que aquí se presenta.

```
int fibonacci(int n)
{
    vector<int> T(1+n);
    T[1] = 1; T[2] = 1;
    for (int i=3; i<=n; i++) {
        T[i] = T[i-1] + T[i-2];
    }
    return T[n];
}
```

Algoritmo 6.3 *Cálculo de los números de Fibonacci.*

Como ejemplo de la técnica, del Algoritmo 6.3 podemos fotografiar tres características fisiológicas. Primera, el vector dinámico ocupa un espacio $\Theta(n)$. Segunda, el tamaño del subproblema solucionado va creciendo. Tercera, se retorna $T[n]$. Estas propiedades son determinantes de la programación dinámica.

Aún así, en el Algoritmo 6.3 no se resuelve ningún problema de optimización. En este sentido, se aparta de la metodología.

La eficiencia temporal del Algoritmo 6.3 pertenece a $\Theta(n)$ y por tanto a $O(n)$. La eficiencia espacial, que en este capítulo adquiere relevancia, es también $\Theta(n)$, o sea, $\Omega(n)$. Recordad que cuando hablamos de tiempo nos interesa saber cuánto tardará como máximo, y por eso ponemos la O. En cambio cuando hablamos de espacio, pensamos en cuánto se necesitará como mínimo, y de ahí la Ω.

Mejora Sistemática

Ahora atención. En el Algoritmo 6.3, se puede comprobar que los valores anteriores de la función utilizados para calcular el actual es un número fijo, relativo al actual. Siempre accedemos al último y al penúltimo número calculados. Eso significa, con los ojos cerrados e independientemente del algoritmo que se trate, que la eficiencia espacial puede ser mejorada, sustituyendo la tabla entera por los únicos dos valores que se utilizan en cada iteración. El cálculo del enésimo número de Fibonacci puede ser implementado con un algoritmo tan simple como el que figura en el Algoritmo 6.4. Esta nueva versión continua con una eficiencia temporal de $\Theta(n)$. La espacial, en cambio, se ha reducido a $\Theta(1)$.

```
int fibonacci_eficient(int n)
{
    int a = 1;
    int b = 1;
    for (int i=3; i<=n; i=i+2) {
        a = a + b;
        b = a + b;
    }
    return (n%2) ? a : b;
}
```

Algoritmo 6.4 *Mejor eficiencia espacial para el cálculo de los números de Fibonacci.*

Por otro lado, en el Algoritmo 6.4 se dejan de utilizar algunas características paradigmáticas de la programación dinámica. Ya no se utiliza ninguna tabla cuyo tamaño sea función de la entrada, ni, está claro, tampoco se retorna *T[n]*. Que un algoritmo se ajuste a la metodología de la programación dinámica no lo compromete a ninguna estructura ni ningún valor de retorno. Sólo faltaría. Participar del esquema de la programación dinámica significa moverse por soluciones óptimas de los subproblemas que van creciendo. En este sentido, el Algoritmo 6.4 respeta totalmente la filosofía.

6.4 Esquema Algorítmico de Programación Dinámica

En el Esquema 6.1 se muestra el esquema algorítmico de la programación dinámica. Es un esquema iterativo.

```
algoritmo programacion_dinamica(problema)
{
    T[1] = rellenar_casos_triviales(problema)
    para i=2 hasta n hacer
        T[i] = solucionar_desde_1_hasta(i,problema)
    fpara
    retorna T[n]
}
```

Esquema 6.1 *Programación Dinámica.*

Los problemas que se resuelven mediante programación dinámica parten de una recurrencia, una ecuación en diferencias. Comienzan solucionando los casos triviales de la recurrencia. Estas soluciones casi siempre son simples asignaciones. Después, iterativamente, van calculando soluciones a problemas de tamaños crecientes hasta conseguir el tamaño de la instancia planteada originalmente.

Desde una óptica más filosófica, la característica identitaria de esta metodología es, por decirlo de alguna manera, una *naturaleza acumulativa*. La cosa más importante del esquema es que en cada paso del bucle, tenemos en cuenta un paso más, adicional. Lo que se hace en cada iteración no sólo depende de la iteración en cuestión, sinó de lo que se ha hecho desde la primera hasta la actual. Eso recuerda a la integración de funciones en el anásilis matemático. Naturaleza acumulativa.

De los problemas que se presentan a continuación, el primero, el del número de subconjuntos, no es un problema de optimización, igual que tampoco lo era el de Fibonacci. La razón por la cuál se estudian estos problemas en programación dinámica es simple. Porque respetan minuciosamente la estructura del esquema algorítmico. Aún así, no son problemas de optimización. O sea, alguna diferencia fisiológica deben tener respeto a los demás. Y efectivamente así es. A diferencia de los problemas de Fibonacci o del número de subconjuntos, los problemas típicos de la programación dinámica, asignan el valor de la tabla en la iteración actual dentro de una sentencia alternativa. Es decir, en el Esquema 6.1 se hubiera podido poner directamente que la función $solucionar\_desde\_1\_hasta(i, problema)$ consiste en un máximo o un mínimo.

Así pues, como característica fisiológica adicional a las tres citadas en el Algoritmo 6.3 con el ejemplo de Fibonacci, tenemos que el cálculo de $T[i]$, en el

Esquema 6.1 acostumbra a tener un aspecto como

$$T[i] = max\ \{T[k], k = 1, \ldots, i-1\}.$$

Además, hay una transformación de sistemas de enumeración. Lo que diferencia el contenido de una posición de la tabla y otra se puede cuantificar en alguna unidad que viene definida en el problema, euros, quilos, metros... en fin, las unidades de la función objetivo.

En el caso paticular de dos variables, las acostumbramos a llamar n y k. Tendremos matrices bidimensionales. O sea, dos bucles. Uno indexado por i, correrá para $i = 1$ hasta n. El otro bucle, indexado por j para $j = 1$ hasta k, será recorrido íntegramente en cada iteración i del bucle principal. Cada columna de la tabla contendrá el mejor valor obtenido para resolver el problema planteado con tamaño i, y habiendo tomado las decisiones desde 1 hasta k. Está claro que ya que siempre tendremos la posibilidad de tomar la última decisión negativamente, rechazando el último candidato, además de que acostumbraremos a usar expresiones del tipo

$$T[i][k] = max\ \{T[i][j], j = 1, \ldots, k-1\},$$

también ocurrirá normalmente que una de las opciones del máximo consista en prescindir del candidato actual, j. Así pues, a riesgo de resultar un esquema excesivamente particularizado para algunos problemas, nos atrevemos a concretar, sin ningún tipo de rigor y a nivel orientativo, que la parte interior del bucle tendrá un aspecto como

$$T[i][j] \ = \ max\ \{T[i][j-1], T[i][j-\alpha] + \beta\}$$

Donde α y β vienen con los datos de la instancia. Es decir, el nuevo valor de la función objetivo, $T[i][j]$, para la tamaño actual, i, y con la opción adicional, j, es como mínimo igual al mismo valor que tendríamos rechazando esa opción, $T[i][j-1]$. La otra posibilidad del máximo, $T[i][j-\alpha] + \beta$, es la que consiste en utilizar la opción j en la nueva solución.

Con todo, todavía vamos más allá. Dominar la técnica de la programación dinámica consiste en entender que si somos capaces de definir con todo el rigor el significado de cada uno de los índices de la matriz que se utiliza para un problema, entonces el problema ya está resuelto. Estas definiciones acostumbran a tener un aspecto como

T[i][j] significa el mejor valor obtenido con un problema de tamaño i utilizando las decisiones de 1 a j.

Así pues, si nos dicen en el enunciado del problema cuál es la semántica de cada uno de los índices que llenan la tabla, nos están resolviendo el problema

directamente, y entonces implementarlo se convierte en una tarea trivial. La única gracia que tiene la programación dinámica es definir, con todo el rigor necesario, el sentido de los índices de la tabla que se rellena con valores óptimos parciales, sin confundir preposiciones diciendo *por* en lugar de *desde*. Por cierto, siempre hay que usar la preposición *desde* en esta definición.

Ahora pues, todavía se ven más claras las carencias que tienen, Fibonacci o el número de subconjuntos, como algoritmos de esta técnica. No ilustran la filosofía de la programación dinámica, sinó tan solo la implementación, ya que el significado de los índices nos los da el mismo enunciado del problema.

6.5 Algunos Problemas

Seguidamente se presentan cuatro problemas. El número de subconjuntos posibles de un conjunto de n elementos se denomina también el coeficiente binomial. Es un problema resuelto del que la única cosa que hay que hacer es la implementación. De todas maneras se profundiza en el tema porque resulta misterioso el contenido numerológico que hay detrás. Los siguientes tres problemas son paradigmas de la metodología que nos ocupa. Se pretende mostrar que las analogías en los procedimientos de resolución son muchas. Los tres problemas se resuelven exactamente igual, casi. El primero, la máquina de volver cambio, hace el papel de problema puente entre los más fáciles como Fibonacci o el coeficiente binomial, y los más difíciles, la mochila, y el algoritmo de Floyd. Estos dos últimos se muestran también para dar implementaciones satisfactorias a problemas que con la estrategia voraz nos habíamos quedado a medias.

Hay problemas que se han convertido en referencia. Estamos estudiando métodos de resolución de problemas, y es por eso interesante analizar cómo se soluciona un mismo problema con diferentes técnicas.

6.5.1 Número de Subconjuntos

O Coeficiente Binomial. En esta sección se aborda el problema de cuántos subconjuntos se pueden hacer de un conjunto de n elementos. La solución es 2^n. ¿Qué curioso, no? Y... ¿Cómo es que hay un dos en la fórmula?. ¿Qué tiene de mágico el número dos que a la hora de expresar un concepto tan básico como el número de subconjuntos de un conjunto de n elementos, aparezca en la fórmula?. Pues muy sencillo. El número dos está en el origen de la definición de subconjunto. La cosa es que no podemos hacer un subconjunto sin hacer dos. Y eso justifica su presencia en la fórmula. El número de subconjuntos que puedo hacer a partir de un conjunto de n elementos es 2^n. Claro, eso incluye como subconjuntos el conjunto vacío, los n subconjuntos de un solo elemento, el número de subconjuntos de dos elementos, cuántos de tres, de cuatro..., y así hasta n. Hagamos una cosa, nos definimos una función a la que le demos dos

valores enteros y nos devuelva otro valor entero. Para utilizarla, al operador le denominamos *sobre*, igual que al operador de la suma le llamamos *más*. Así como decimos tres más dos, y nos referimos al resultado de la suma, podemos decir tres sobre dos, y nos referimos al resultado de esta función que estamos definiendo. La denotaremos de forma rara, con paréntesis grandes, y el primer argumento arriba y el segundo abajo. Definimos la función como

$\binom{n}{k}$: Cuántos subconjuntos de k elementos se puede hacer con n elementos.

Así pues, esta función sólo está definida para $0 \leq k \leq n$, un dominio triangular.

Es bien conocida la expresión analítica que resuelve $\binom{n}{k}$. Tiene la forma

$$\binom{n}{k} = \frac{n!}{(n-k)!\ k!} \tag{6.2}$$

aunque ciertamente duele un poco en los ojos. De hecho, no es una expresión canónica en el sentido que de alguna manera se debería poder simplicar. Observad que en el cociente de la derecha de la fórmula la parte de $(n-k)!$ del denominador sistemáticamente se podrá tachar con la parte correspondiente del $n!$ del numerador. De hecho, esta fórmula es más fácil de recordar verbalmente que gráficamente. Por ejemplo, cinco sobre dos es cinco por cuatro, dividido entre dos por uno. Es más fácil de decir que de escribir. En la lotería primitiva, hay que adivinar seis números entre cincuenta. O sea, la probabilidad de que os toque es de una entre 15890700 = (50*49*48*47*46*45)/(6*5*4*3*2*1). Es fácil si interpretamos que la k de la operación nos dice cuántos términos hay que multiplicar del factorial de n en el numerador, y después, dividir este producto por $k!$. Sin duda más fácil de recordar verbalmente que mirando la fórmula.

Y todavía hay una manera más sencilla de calcular todas las posibilidades de n's sobre k's. Es decir, para toda n y para toda k. Para usar esta manera, sin embargo, son necesarios papel y lápiz, y dibujar el triángulo de Tartaglia, también llamado de Pascal. Lo empezamos por el vértice superior, y lo vamos expandiendo hacia abajo hasta donde se precise. Un procedimiento totalmente propio de la programación dinámica. En el vértice superior, que viene a ser la fila cero, ponemos un 1. Entonces en la fila siguiente, dos unos, un 1 a la izquierda del de encima, y otro a la derecha. Cada nueva fila empieza con el número 1, y a partir de ahí, en cada posición se pone la suma de los dos números que haya a izquierda y derecha de la fila anterior. En la Figura 6.2 se pueden ver las quince primeras filas de este triángulo.

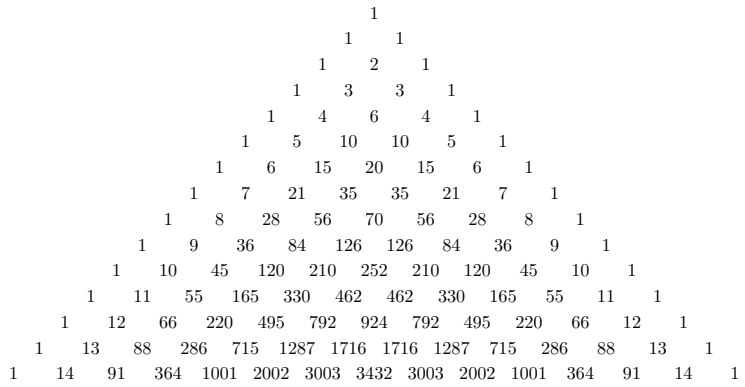


Figura 6.2: *Triángulo de Tartaglia, o de Pascal.*

En el triángulo de la Figura 6.2 tenemos desplegados todos los coeficientes binomiales, para cualquier pareja de números n, k, con $0 \leq k \leq n \leq 14$. La utilidad que tiene este triángulo es que, si tenemos un conjunto de n elementos, podemos saber rápidamente cuantos subconjuntos de cada cardinalidad inferior o igual a n podemos extraer. Y claro, el número total de subconjuntos es la suma de los subconjuntos de cero elementos, más los de un solo elemento, más los de dos, etcétera, y por tanto, la suma de toda la fila del número n en cuestión.

Por ejemplo, en la Figura 6.3 nos concentramos en la fila del recuadro, correspondiente a $n = 6$ elementos. La información que nos da el triángulo es la siguiente.

Con un conjunto de 6 elementos, que podemos llamar a, b, c, d, e, f podemos obtener 1 subconjunto de cero elementos, el conjunto vacío, que es subconjunto de todos los conjuntos y es por eso que todo el triángulo empieza con unos en la primera posición de todas las filas. También podemos obtener 6 conjuntos de un solo elemento. Denotémoslos con *(a), (b), (c), (d), (e),* y *(f).* O podemos hacer subconjuntos de dos elementos. Serán *(a,b), (a,c), (a,d), (a,e), (a,f)*, entonces *(b,c), (b,d), (b,e)* y *(b,f)*, y también *(c,d), (c,e)* y *(c,f).* Y aún, *(d,e)* y *(d,f)*, y además, *(e,f).* En total son 15, como dice el triángulo.

De subconjuntos de tres elementos podemos hacer 20. Son *(a,b,c), (a,b,d), (a,b,e), (a,b,f), (a,c,d), (a,c,e), (a,c,f), (a,d,e), (a,d,f), (a,e,f), y también (b,c,d), (b,c,e), (b,c,f), (b,d,e), (b,d,f),* y *(b,e,f).* Entonces, *(c,d,e), (c,d,f),* y *(d,e,f).* De subconjuntos de cuatro elementos podemos hacer tantos como de dos, ya que cada vez que hacemos un subconjunto de dos elementos, los que no cogemos forman el subconjunto de cuatro elementos correspondiente. O sea, 15. De hecho, por este motivo el triángulo es simétrico respeto al eje vertical. Porque igual que con los de cuatro elementos, de subconjuntos de cinco elementos también podemos hacer 6, o sea, todos menos el primero, todos menos el segundo, etcétera. Finalmente, de subconjuntos de todos los elementos, sólo podemos hacer 1, que también es el complementario del conjunto vacío, o sea,

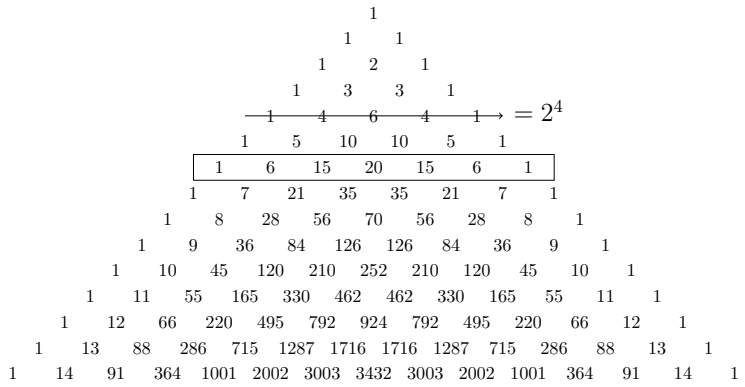


Figura 6.3: *Utilitado del triángulo.*

del 1 del otro extremo de la fila.

Así pues, el triángulo de Tartaglia es una herramienta útil y amable para desglosarnos los subconjuntos que podemos hacer de un conjunto. Como se ha dicho al principio de esta sección, el número de subconjuntos posibles de un conjunto de n elementos es 2^n. Por tanto, la suma de todos los términos de una fila cualquiera es dos elevado al número de fila en cuestión. En la Figura 6.3 se explicita la suma de los elementos de la quinta fila, para $n = 4$, que suman $1 + 4 + 6 + 4 + 1 = 2^4 = 16$.

Con todo, este triángulo tiene cosas de una belleza incuestionable. Por ejemplo, si pintamos todos los números pares del triángulo de Tartaglia, nos aparece una ornamentación geométrica como la mostrada en la Figura 6.4.

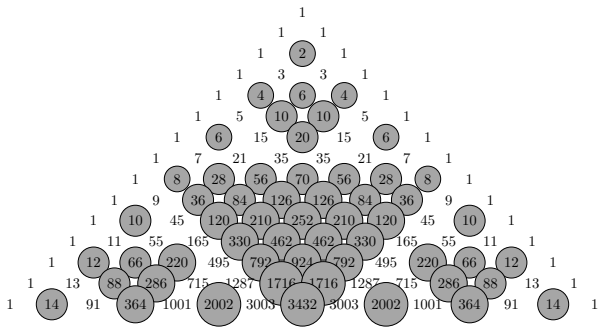


Figura 6.4: *Números pares en el triángulo de Tartaglia.*

Este patrón va aumentando su triángulo central al tiempo que van apareciendo triángulos menores cuando hacemos la representación sobre n's mayores. Otra curiosidad es el aspecto que toma al pintar los múltiplos de 5. Se muestra en la Figura 6.5.

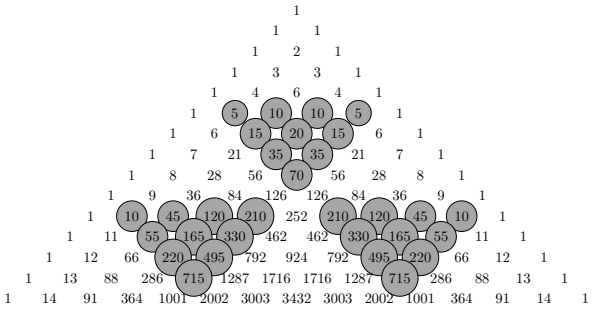


Figura 6.5: *Múltiplos de* 5 *en el triángulo de Tartaglia.*

Bien, una vez presentados tanto el instrumento como el procedimiento para calcularlo, echamos un vistazo a la implementación del algoritmo.

Gráficamente, resulta de sentido común implementar el triángulo de Tartaglia en una matriz triangular. O sea, utilizando la mitad inferior izquierda, por ejemplo, de una matriz. Eso es, sólo rellenamos los valores de la matriz para los que la columna es inferior a la fila. Tanto en la primera columna de la matriz como en la diagonal haremos las asignaciones correspondientes a los casos triviales. Después, entraremos en un bucle que irá llenando la matriz, calculando los nuevos valores en función de los ya calculados y rellenando así nuevas posiciones.

En el Algoritmo 6.5 se resuelve mediante la estrategia de la programación dinámica el cálculo de $\binom{n}{k}$.

```
int subconjuntos(int n, int k)
{
    n++;
    matriz<int> c(1+n,1+n,0);
    c[1][1] = 1;

    for (int i=2; i<=n; i++) {
        c[i][1] = 1;
        c[i][i] = 1;
        for (int j=2; j<i; j++) {
            c[i][j] = c[i-1][j-1] + c[i-1][j];
        }
    }
    return c[n][k+1];
}
```

Algoritmo 6.5 *Algoritmo para el cálculo del coeficiente binomial de* n *sobre* k.

Como se ve, el algoritmo para el cálculo del número de subconjuntos de un conjunto de n elementos, que se pueden hacer con k elementos, recibe como parámetros de entrada los dos argumentos n y k. Un código más fino debería verificar que $k \leq n$.

Precisamente, en este caso va de maravilla que los índices comiencen a contar en cero. Conviene indexar la primera fila del triángulo con el índice 0. Por esto el Algoritmo 6.5 comienza incrementando el valor de n. Y continua con el cálculo tal y como ha sido descrito.

Tanto la eficiencia espacial como la temporal del problema del coeficiente binomial resuelto con programación dinámica es $\Theta(n^2)$. Es característica de esta metodología la equivalencia entre eficiencias, ya que si nos guardamos los resultados intermedios en algún espacio, es precisamente para reducir el tiempo de cálculo de estos resultados de manera que la eficiencia temporal venga dominada por el hecho de llenar cada posición de las tablas.

Por otra parte, es importante que quede claro que el Algoritmo 6.5 deja mucho que desear. De entrada, podemos utilizar la mejora sistemática mencionada en la Sección 6.3.1. Esta mejora de eficiencia espacial, como ya se ha dicho, es algo que siempre conviene comprobar. Cuando en la expresión recursiva se pueda saber exactamente cuáles elementos previamente calculados hacen falta para el actual, se puede ahorrar el espacio que se dedica a los que no se precisan.

```
int subconjuntos_eficiente(int n, int k)
{
    n++;
    vector<int> c(1+n); // fila actual.
    vector<int> cc(1+n); // fila anterior.
    c[1] = 1;

    for (int i=1; i<=n; i++) {
        c[1] = 1;
        for (int j=2; j<i; j++) {
            c[j] = cc[j-1] + cc[j];
        }
        c[i] = 1;
        for (j=1; j<=i; j++) cc[j] = c[j];
    }
    return c[1+k];
}
```

Algoritmo 6.6 *Mejor eficiencia espacial para el cálculo de* n *sobre* k.

En el Algoritmo 6.6 se puede observar la mejora. Hemos reducido la eficiencia espacial del algoritmo a $\Omega(n)$. Con un código un poco más astuto se podría

evitar la copia del vector actual al anterior del final del bucle, y trabajar alternadamente intercambiando los rols entre los dos vectores. La eficiencia temporal del Algoritmo 6.6 no es mejor que la del Algoritmo 6.5. Es igual, $\Theta(n^2)$.

Si queremos ir más allá, está claro que también se podría resolver el problema utilizando la definición (6.2). Es decir,

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

que aunque sea trivial, también se muestra en el Algoritmo 6.7. En esta última versión se esquiva el inconveniente de que al hacer el producto de los k términos $n*(n-1)*(n-2)*\cdots*(n-(k-1))$ se desborde la capacidad para guardar números enteros, o sea, que el número que calculamos supere $2^{32}-1$. Por esta razón, lo que se hace es multiplicar, dividir, multiplicar, dividir,... y así se mantiene más estable el valor de los cálculos intermedios. Por ejemplo, para el caso de la lotería primitiva, en lugar de calcular $(50*49*48*\cdots*45)/(6*5*\cdots*1)$, calculamos $(((50/6)*49)/5)*48\ldots$

El Algoritmo 6.7 muestra la mejor implementación para el cálculo del coeficiente binomial. Este método ya no tiene nada que ver con la programación dinámica, aún así, para resolver este problema es el más eficiente de los que se han mostrado.

```
int subconjuntos_mas_eficiente(int n, int k)
{
    int num = 1;
    for (int i=1; i<=k; i++) {
        num = num * (n+1-i)/i;
    }
    return num;
}
```

Algoritmo 6.7 *Eficiencias óptimas para el cálculo de* n *sobre* k.

Es fácil ver que el Algoritmo 6.7 tiene una eficiencia temporal de $\Theta(n)$, y espacial de $\Theta(1)$.

6.5.2 El Problema de Devolver Cambio

Del esquema de programación dinámica, los dos problemas vistos hasta ahora tan solo tenían la forma. No se puede decir que tuviesen una función objetivo, ni

que pretendiesen maximizar ni minimizar valor alguno. Ahora nos enfrentamos con un tipo de problema más completo.

Definición 6.1 Problema de Devolver Cambio. *Disponiendo de un sistema monetario con monedas de n valores diferentes,* $1 = v_1 < v_2 < \cdots < v_n$*, encontrar el mínimo número de monedas para devolver un cambio de cierta cantidad* K*.*

Observad que que v_1 sea igual a 1 evita problemas de no factibilidad. En todos los casos se podrá devolver cualquier cambio con alguna cantidad de monedas, cosa que no ocurre en muchos cajeros automáticos en los que no se puede extraer 30 euros porque solamente disponen de billetes de 20 y 50.

Aproximación con algoritmos voraces

Está claro que el procedimiento utilizado por cualquier vendedor o vendedora para resolver este problema es óptimo. Se trata de un algoritmo voraz caracterizado por utilizar el criterio de selección voraz siguiente.

Devolver siempre la moneda de valor mayor que se pueda.

No obstante, este algoritmo sólo da el número de monedas óptimo para según qué sistemas monetarios. Para el actual que se utiliza en Europa, funciona. Si tenemos un sistema formado por monedas de $n = 3$ valores diferentes, $v_1 = 1$ céntimo, $v_2 = 2$ céntimos, y $v_3 = 5$ céntimos, por ejemplo, entonces el algoritmo voraz nos proporciona exactamente el valor óptimo. Es decir, el mínimo número de monedas necesarias para retornar cualquier cambio. Sin embargo, eso sólo será así siempre que $v_i \geq 2 * v_{i-1}$, $\forall i = 2, \ldots, n$.

Ahora imaginemos un sistema monetario formado por monedas de $n = 3$ valores, $v_1 = 1$ céntimo, $v_2 = 4$ céntimos, y $v_3 = 6$ céntimos. Además, suponemos que se trata de devolver un cambio de $K = 8$ céntimos. Entonces el algoritmo voraz nos daría un resultado de tres monedas, $6 + 1 + 1$. El valor óptimo, en cambio es dos, $4 + 4$.

Quizás se puede tildar este problema de frívolo. Realmente, sería muy extraño que un sistema monetario fuera tan complicado. Da igual, una vez más conviene poner la atención en la naturaleza del problema y en el método de resolución, más que en la utilidad concreta de lo que se resuelve.

Aproximación con programación dinámica

De la resolución del problema de devolver cambio con la estrategia de la programación dinámica, la idea brillante es la de siempre en programación dinámica:

Definir semánticamente los índices de la tabla que nos disponemos a construir. Con esta definición está todo dicho. Después nos interesaremos por los detalles.

$C[i][j]$ contendrá el mínimo número de monedas necesarias para retornar un cambio de j céntimos utilizando monedas con valores desde v_1 hasta a v_i.

Que quede claro, en programación dinámica el contenido de la tabla contiene directamente valores correspondientes a los de la función objetivo, o soluciones a subproblemas. Eso es así porque como se ha ido diciendo, esta técnica resuelve problemas pequeños óptimamente para después, gracias al principio de optimalidad, hacer crecer la bola de nieve.

Desde una perspectiva más filosófica, observad también que con el esquema algorítmico de la programación dinámica es fácil decir cuántos, y no tan fácil decir cuáles. O sea, respondemos el mínimo número de monedas necesario, pero no cuántas monedas de cada tipo.

Volviendo al tema. Un vez descrita, con todo rigor y sin confundir preposicions, la definición semántica de los índices, ya podemos descansar. El problema ya está prácticamente resuelto. Ahora abordamos sistemáticamente un par de cuestiones.

- ¿Dónde quedará el resultado?
  - En $C[n][K]$. Normal. Eso tiene buena pinta.
- ¿Cuáles son los casos triviales?
  - $C[i][0] = 0$, $\forall i \in \{0, \ldots, n\}$. Para retornar 0 céntimos de cambio, el número mínimo de monedas es 0, independientemente de los valores disponibles.
  - $C[1][j] = j$, $\forall j \in \{1, \ldots, K\}$. Si tan solo disponemos de monedas del primer tipo, que como se ha dicho se supone $v_1 = 1$, necesitamos j para poder retornar un cambio de j céntimos.

Llegados a este punto, cogemos papel y lápiz y nos ponemos a escribir los valores de la matriz tal como nos dicte el sentido común. Disponiendo de los casos triviales, seguimos con la segunda columna o fila, da igual, y para cada posición nos vamos preguntando cuál es el contenido que le corresponde. Vamos haciendo esto hasta que nos cansemos, o hasta que seamos capaces de observar qué regla estamos utilizando. Y de escribirla en forma de recurrencia.

La matriz resultante se muestra en la Figura 6.6, que después de haber escrito parte de su contenido, uno ya se da cuenta de que está utilizando la fórmula que se indica también en la figura. Aclaremos el valor de la posición de la quinta columna, $K = 4$, tercera fila, $v_3 = 6$, por ejemplo. El valor es 1. Eso significa que para devolver cuatro céntimos utilizando monedas de 1, 4, o 6 céntimos, el mínimo número de monedas es 1.

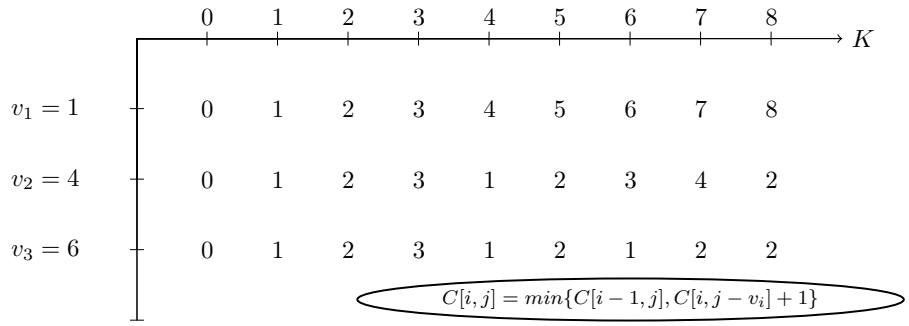


Figura 6.6: *Recurrencia para el problema de devolver cambio.*

En general, podemos analizar la semántica de la recurrencia utilizada.

$$C[i,j] \;=\; min\;\{C[i-1,j], C[i,j-v_i]+1\} \tag{6.3}$$

Ya se ha mencionado, pero conviene repetir, que con la definición (6.3), se calcula el mínimo número de monedas necesarias para devolver un cambio de j céntimos utilizando monedas de valores $v_1, v_2, \ldots, v_i$. Lo que se hace ahora recuerda la inducción matemática. El razonamiento comienza a partir de la pregunta:

Disponer de un nuevo tipo de moneda, i, de valor v_i, sirve para reducir el número de monedas a devolver de cambio, para retornar los j céntimos?

La resposta es una disyuntiva. Pueden pasar dos cosas.

- El nuevo tipo de moneda, de valor v_i, no nos sirve para reducir la cantidad de monedas. En otras palabras, no utilizamos ninguna moneda del nuevo valor. Entonces, el número de monedas mínimo es igual al que teníamos sin utilizar el nuevo tipo, para retornar la misma cantidad de cambio, $C[i-1,j]$.
- El nuevo tipo de moneda, de valor v_i, efectivamente nos sirve para reducir la cantidad de monedas. En otras palabras, utilizamos una moneda del nuevo tipo para reduir el número de monedas total. Entonces, el número de monedas mínimo necesario es el mismo que teníamos para devolver una cantidad igual al cambio actual, sin usar la nueva moneda, menos el valor de la nueva moneda, más uno. O sea, $C[i-1,j-v_i]+1$. Si además tenemos en cuenta que la nueva moneda puede usarse varias veces, tenemos $C[i,j-v_i]+1$.

La solución óptima será lógicamente la que dé el valor mínimo de la decisión que se plantea, tal como dice la expresión (6.3).

Del problema de devolver cambio tan solo queda por presentar la implementación de la resolución. En el Algoritmo 6.8 se puede ver el código de esta implementación.

```
int devolver_cambio(vector<int> v, int K)
{
    int n = v.n-1;
    matriz<int> C(1+n,1+K,0);
    for (int j=1; j<=K; j++) {
        C[1][j] = j;
        for (int i=2; i<=n; i++) {
            C[i][j] = C[i-1][j];
            if (j>=v[i]) {
                if (C[i][j] > C[i][j-v[i]] + 1) {
                    C[i][j] = C[i][j-v[i]] + 1;
                }
            }
        }
    }
    return C[n][K];
}
```

Algoritmo 6.8 *Algoritmo para el problema de devolver cambio.*

En un análisis breve de este algoritmo observamos fácilmente que...

- en la cabecera se ve que la rutina recibe el vector con los valores de las monedas, v, que como precondición se suponen ordenados por orden creciente, y la cantidad de cambio a retornar, K.
- una vez dentro del cuerpo del procedimiento en la declaración de la matriz se rellena de ceros para iniciar la primera columna. En la primera línea del interior del bucle más externo se inicializan los otros casos triviales.
- en el corazón de los bucles, la primera sentencia alternativa, *if (j>=v[i])*, mantiene los índices de la matriz en sus dominios. El segundo *if*, es el que sirve para buscar el mínimo.
- el valor de retorno de la rutina es el valor de la solución.

Más genéticamente, la tamaño de la matriz C viene dado por el tamaño de la instancia, como siempre pasa en la programación dinámica. Y más cosas de genética. Por fin, nos encontramos con una función objetivo como diós manda. En el interior de los bucles, cuando se llena la tabla, hay un cálculo de un mínimo. Eso ya son los cromosomas. Este es un aspecto fisiológico que todavía no nos

habíamos encontrado en ninguno de los problemas anteriores de este capítulo, y en el cual reside la naturaleza del problema de devolver cambio como problema de variables enteras.

El algoritmo para el problema de devolver cambio tal y como se ha implementado en esta sección tiene eficiencias temporal y espacial de $\Theta(nK)$. Por otra parte, la mejora sistemática de los problemas precedentes también puede ser implementada en este caso, ya que tan solo es necesaria la fila precedente a la actual para calcular la actual en cada iteración. Es decir, con dos filas ya hacemos. Esta implementación final se deja como ejercicio al lector, que se supone que no tendría demasiada dificultad para realizarla. Entonces, la eficiencia espacial quedaría reducida a $\Theta(2K)$, es decir, $\Theta(K)$.

En cualquier caso, se proporciona la solución óptima, cosa que hasta ahora no teníamos manera de conseguir, y que representa un paso de gigante a la hora de resolver problemas de optimización con variables enteras.

6.5.3 Problema de la Mochila

Recordemos la definición del problema de la mochila, ya dada en la Sección 5.3.

Definición 6.2 Problema de la Mochilla. *De entre n objetos que tienen pesos $w_i \in \mathbb{R}$, para $i = \{1, \ldots, n\}$, y valores $v_i \in \mathbb{R}$, para $i = \{1, \ldots, n\}$, conseguir el máximo valor posible, siempre que el peso total no supere la capacidad de peso W de la mochila.*

La resolución es análoga al problema anterior. Llenaremos una matriz $V[i][j]$ definiendo los índices como sigue.

$V[i][j]$ contendrá el máximo valor obtenible con una mochila capaz de cargar j kilos, utilizando tan solo objetos de los tipos desde 1 hasta i.

Idea expuesta. Una vez definidos con esta precisión los índices de la matriz, ya tenemos el problema resuelto. Ahora, los detalles. Procedemos dando respuesta a las cuestiones siguientes.

- ¿Dónde quedará el resultado?
  - En $V[n][W]$, bonito.
- ¿Cuáles son los casos triviales?
  - $V[i][0] = 0$, $\forall i \in \{0, \ldots, k\}$. Si la mochila no puede llevar ningún peso, no podemos conseguir ningún valor, independientemente de los objetos disponibles.

– $V[1][j] = v_1 * j/w_1$, $\forall j \in \{1, \ldots, n\}$. Eso es la división entera. Si tan solo podemos cargar objetos del primer tipo, podremos conseguir un valor igual al número de objetos enteros de ese tipo que quepan en la mochila, por el valor de cada uno de ellos.

Establecidas estas ideas, continuamos resolviendo el problema. Escribimos ahora la matriz de alguna instancia concreta en papel y lápiz. Con algún ejemplo calculamos los términos que hagan falta hasta podernos abstraer y deducir el término genérico. Suponemos una instancia en la que hay cinco objetos con pesos $w = (1, 2, 5, 6, 7)$, y valores $v = (2, 6, 18, 22, 28)$. Para este ejemplo, la representación matricial se puede observar en la Figura 6.7. También en la misma figura aparece la conclusión que hay que deducir de la observación del proceso anterior.

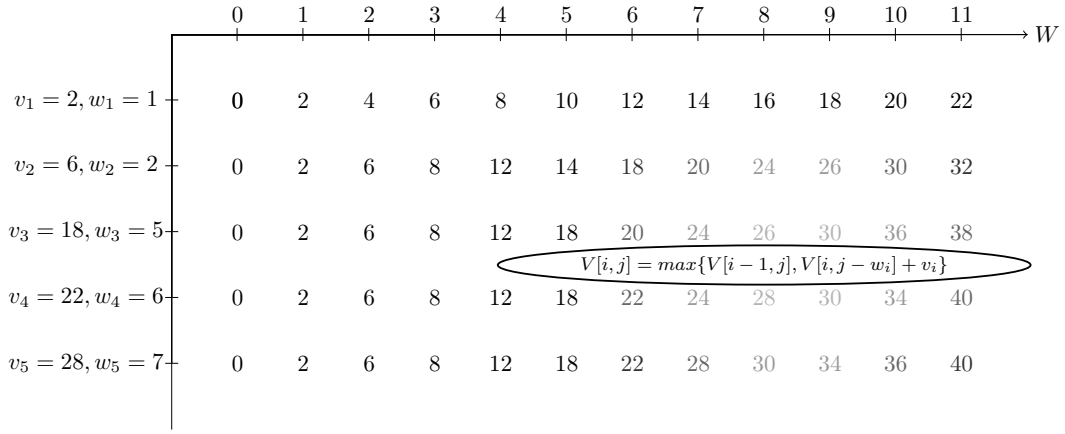


Figura 6.7: *Recurrencia para el problema de la mochila.*

El mismo análisis semántico que se ha hecho en la sección anterior se puede hacer para el caso de la mochila. Centramos el discurso en la recurrencia

$$V[i,j] \;=\; max\ \{V[i-1,j], V[i,j-w_i] + v_i\} \tag{6.4}$$

Suponiendo que los valores óptimos, o sea máximos, que se pueden obtener para los índices menores a los actuales están correctamente calculados en la matriz V, vamos a ver qué hacemos cuando aparece un nuevo objeto disponible. Es importante observar que siempre suponemos que el problema augmenta en la dirección de la variable acumulativa.

¿Qué hacemos con un nuevo tipo de objetos? Una de dos,

- Lo rechazamos. O sea, el máximo valor que obtenemos es el mismo que cuando no cargábamos ningún objeto de este nuevo tipo i en la mochila $V[i-1, j]$.

- Lo cargamos, entonces, el valor obtenido aumenta en v_i. Y por otra parte, la capacidad de la mochila se ve reducida en w_i. En definitiva, el nuevo valor obtenido se puede dar en base al valor obtenido con una mochila con la capacidad reducida, $j - w_i$. Eso es, un valor igual a $V[i, j - w_i] + v_i$.

Otra vez, la mejor solución será la que maximice la función objetivo, tal como se ha definido en la recurrencia (6.4).

```
double mochila(vector<double> w, vector<double> v, double W)
{
      matriz<double> V(1+n,1+W,0);

      for (int j=1; j<=W; j++) V[1][j] = (int) v[1]*(j/w[1]);

      for (int i=2; i<=n; i++) {
           for (int j=1; j<=W; j++) {
               V[i][j] = V[i-1][j];
               int vi = (int) v[ii];
               int wi = (int) w[ii];
               if (j>=wi) {
                     if (V[i][j] < V[i][j-wi] + vi) {
                          V[i][j] = V[i][j-wi] + vi;
                     }
               }
           }
      }
      return V[n][W];
}
```

Algoritmo 6.9 *Algoritmo de programación dinámica para el problema de la mochila.*

El Algoritmo 6.9 recibe como parámetros de entrada los vectores de pesos y valores, w y v, y la capacidad de la mochila, W. El valor de retorno de la función es la solución del problema.

Declaramos la matriz de dimensiones iguales al tamaño de la instancia, como es propio de esta técnica, y la inicializamos a cero, para resolver la primera columna. Hay un bucle específico para el caso trivial de la primera fila, $\Theta(W)$. Lo que sigue no es más que la implementación directa de la recurrencia explicada más arriba.

Las eficiencias del Algoritmo 6.9 son $\Theta(nW)$, tanto la espacial como la temporal. En lugar de mostrar la implementación utilitzando la mejora sistemática,

como antes se deja la implementación del Algoritmo 6.9, aunque de todas formas, es importante darse cuenta de que una vez más se podría prescindir de la matriz y utilizar tan solo las dos últimas filas en cada iteración. Con esta mejora, la eficiencia espacial quedaría reducida a $\Theta(W)$, como antes.

6.5.4 Algoritmo de Floyd


Robert Floyd

Robert Floyd (1936-2001) fue un ingeniero informático norteamericano. Un pionero de la ingeniería de la computación que dedicó gran parte de sus esfuerzos en matematizar la programación informática. Interesado en la verificación formal, su principal contribución fue la propuesta del método de las invariantes definiéndolas como aserciones lógicas asociadas a ciertos puntos del código algorítmico. Compañero de Donald Knuth, tuvo una participación intensa en la obra *The Art of Computer Programming*, de cinco volúmenes y todavía no acabada hoy día. Es más, se planifica acabar el año 2015. De lenguajes de programación, Donald Knuth decía que sólo había cinco buenos papeles publicados, y cuatro eran de Floyd. A la edad de seis años demostró ser un niño prodigio, y a los catorze había acabado los estudios superiores y entró en la universidad de Chicago para estudiar arte neoliberal, o sea hippy. Más tarde estudió física. Pero por lo que se refiere a los ordenadores, iba bastante a la suya. Un autodidacta nato. Floyd propuso el algoritmo de caminos mínimos que sigue. De todas maneras, el material relativo a la verificación formal tiene sin duda un valor intelectual mayor. Además, le gustaba jugar al backgammon. Y también, como Dijkstra, se jubiló, cosa que hace pensar que la gente que se dedica a estos temas, también les gusta vivir la vida.

En la Sección 5.5.1 se ha expuesto el algoritmo de Dijkstra para al cálculo de caminos mínimos entre un vértice y todos los demás en un grafo no dirigido con costes no negativos. El algoritmo de Floyd es más genérico. No sólo porque nos da las distancias entre cualquier pareja de nodos, sinó también porque admite costes negativos. De paso, detecta ciclos de peso negativo y cuando encuentra un ciclo de peso negativo, el algoritmo se para.

Entonces, ¿Por qué necesitamos el algoritmo de Dijkstra?.

Muy sencillo, porque es más rápido en hacer lo que hace, y necesita menos espacio.

Este algoritmo se definió en grafos dirigidos, y la misma idea se ha utilizado con otras finalidades. Por este motivo, también es conocido por el nombre de Floyd-Warshall. No obstante, el algoritmo de Warshall está orientado a la clausura transitiva de un grafo y no explícitamente a los caminos mínimos. Eso

hace que el grafo sea sin pesos, la operación suma de distancias se transforma con la conjuntiva lógica AND de existencias, y la operación mínimo, en la disyuntiva lógica OR. El mismo método se utiliza para la inversión de matrices con el procedimiento de Gauss Jordan, o en un algoritmo de Kleene relacionado con autómatas finitos.

El algoritmo de Floyd sirve para obtener la matriz de distancias de un grafo con pesos. Es una matriz de $n \times n$ valores. Eso es, de n nodos por n nodos, con las distancias mínimas entre cualquier pareja. Una matriz de adyacencias, vamos. Lo hace en base al principio de optimalidad. La estructura de datos que utiliza es un cubo. El problema se resuelve, en principio, en una matriz de tres dimensiones que, mediante la mejora sistemática podemos reducir a una secuencia temporal de matrices de dos dimensiones.

La resolución es análoga a problemas anteriores. Llenaremos una matriz $D[i][j][k]$ definiendo los índices.

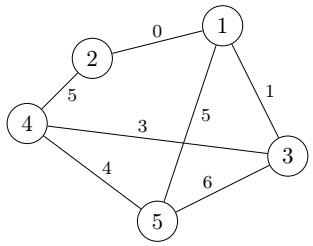
$D[i][j][k]$ contendrá el valor del camino mínimo entre los vértices i y j teniendo en cuenta que este camino sólo puede utilizar como vértices de paso los indexados desde 1 hasta k.

Idea expuesta. Problema resuelto. Ahora, los detalles. Procedemos dando respuesta a las cuestiones siguientes.

- ¿Dónde quedará el resultado?
  - En $D[i][j][n]$.
- ¿Cuáles son los casos triviales?
  - $D[i][j][0] = E$, $\forall i, j \in V$. Si no podemos utilizar ningún vértice adicional, los caminos mínimos entre los nodos estarán formados por las aristas del grafo entre vértices vecinos. Si dos vértices no son vecinos, la distancia inicial será ∞.

Un vez establecido, con todo el rigor, el criterio para llenar la estructura de datos, ya podemos coger papel y lápiz, inventarnos una instancia que nos parezca representativa, y empezar a rellenar la secuencia de matrices que pretendemos calcular hasta que nos demos cuenta del término general que estamos utilizando y podamos describirlo formalmente como una recurrencia. En la Figura 6.8 se muestra un ejemplo. Para agilizar la interpretación, llamamos $D_k(i, j)$ a lo que en lenguaje C sería $D[i][j][k]$.

En la parte superior de la Figura 6.8 se puede ver el grafo, y seguidamente la sucesión de matrices D_k que se corresponden con la definición dada más arriba. Este mismo índice k coincide con la iteración del bucle principal del algoritmo. O sea, que en cada iteración consideraremos un nuevo vértice para poder ser utilizado como vértice de paso en cualquier camino.



$$D_0 = \begin{pmatrix} 0 & 0 & 1 & \infty & 5 \\ 0 & 0 & \infty & 5 & \infty \\ 1 & \infty & 0 & 3 & 6 \\ \infty & 5 & 3 & 0 & 4 \\ 5 & \infty & 6 & 4 & 0 \end{pmatrix} \quad D_1 = \begin{pmatrix} 0 & 0 & 1 & \infty & 5 \\ 0 & 0 & \mathbf{1} & 5 & \mathbf{5} \\ 1 & \mathbf{1} & 0 & 3 & 6 \\ \infty & 5 & 3 & 0 & 4 \\ 5 & \mathbf{5} & 6 & 4 & 0 \end{pmatrix}$$

$$D_2 = \begin{pmatrix} 0 & 0 & 1 & \mathbf{5} & 5 \\ 0 & 0 & 1 & 5 & 5 \\ 1 & 1 & 0 & 3 & 6 \\ \mathbf{5} & 5 & 3 & 0 & 4 \\ 5 & 5 & 6 & 4 & 0 \end{pmatrix} \quad D_3 = \begin{pmatrix} 0 & 0 & 1 & \mathbf{4} & 5 \\ 0 & 0 & 1 & \mathbf{4} & 5 \\ 1 & 1 & 0 & 3 & 6 \\ \mathbf{4} & \mathbf{4} & 3 & 0 & 4 \\ 5 & 5 & 6 & 4 & 0 \end{pmatrix}$$

$$D_k[i,j] \;=\; min\{D_{k-1}[i,j], D_{k-1}[i,k] + D_{k-1}[k,j]\}$$

Figura 6.8: *Recurrencia de Floyd para los caminos mínimos.*

El grafo que se muestra de ejemplo es un grafo no dirigido, o sea que las matrices de distancias son simétricas.

Se ve en la Figura 6.8 que $D_0(i,j)$ es exactamente la matriz de adyacencias del grafo dado, con los valores ∞ para cada pareja de nodos no vecinos. Después viene D_1, la matriz de caminos del grafo que sólo pueden pasar por el vértice 1 como nodo de paso. Eso hace que las distancias entre el 2 y el 3, o entre el 2 y el 5 dejen de ser indefinidas, y pasen a ser 1 y 5, correspondientes a los caminos $2-1-3$ y $2-1-5$. Las distancias reducidas en cada iteración aparecen en negrita en la Figura 6.8. En D_1 todavía hay una distancia indefinida entre los vértices 1 y 4.

Para pasar a D_2 recordamos la naturaleza acumulativa de la programación dinámica, y calculamos los nuevos caminos entre cualquier pareja de nodos, teniendo en cuenta que se pueden usar como vértices adicionales de paso, el 1 o el 2. Eso hace que la distancia entre 1 y 4 exista por fin, por el camino $1-2-4$, aunque valga 5. Como el vértice 2 tiene pocos amigos, ya no podemos reducir ningún otro camino. Por tanto, obtengamos D_3. Poder pasar, además de por el 1 y el 2, por el vértice 3 para calcular los caminos mínimos nos abre las puertas a dos reducciones más. Podemos abreviar las distancias entre 1 y 4, y también entre 2 y 4, si utilizamos el nodo 3 en el camino. Así pues, obtenemos la matriz solución, D_3, con las distancias mínimas entre cualquier pareja de nodos.

A partir de D_3 no se puede reducir ningún camino más, por eso en la Figura 6.8 no se muestran más iteraciones, ya que, a pesar de que el índice k todavía no ha llegado a n, la sentencia alternativa del algoritmo rechazaría cualquier otro cambio.

Respeto al significado de la recurrencia (6.5),

$$D_k[i,j] \;=\; min\;\{D_{k-1}[i,j], D_{k-1}[i,k] + D_{k-1}[k,j]\} \tag{6.5}$$

es como en los casos anteriores.

Tenemos una disyuntiva en la qué, dado un nuevo vértice disponible, k, nos planteamos para cada pareja de nodos i, j, si el camino entre ellos puede ser reducido utilizando este nuevo vértice.

- Si no merece la pena pasar por k, entonces la distancia es igual que cuando este vértice no estaba disponible, $D_{k-1}[i,j]$, o sea la primera opción del máximo de la expresión (6.5).
- Si ciertamente por medio del nuevo vértice k podemos reducir la distancia, entonces lo usaremos, y la nueva distancia entre i y j, que pasa por el vértice k será igual a $D_{k-1}[i,k] + D_{k-1}[k,j]$, o sea la segunda.

Independientemente de la semántica de cada una de las dos opciones, la recurrencia (6.5) considerada como un todo, contiene la esencia del problema, igual que pasaba en los ejercicios de las Seccions 6.5.2 y 6.5.3. El punto clave de la técnica de la programación dinámica está en la imposibilidad de poder predecir, para todos los conjuntos de datos de entrada posibles, cuál de las opciones conseguirá este mínimo. Eso es una decisión de las que componen el problema de optimización. La recurrencia tiene el mismo operador que la función objetivo, y de hecho, es un reflejo fiel de ella.

En el Algoritmo 6.10 se puede contemplar una implementación del método de Floyd. También, como en el resto de problemas de este capítulo, una vez establecidos los significados de los índices para rellenar la matriz, entonces todo viene seguido. El algoritmo no hace más que implementar directamente lo que se ha dicho en el análisis del problema. En la cabecera se proclaman dos parámetros. El grafo con pesos de entrada, g, y la matriz de distancias de salida, d.

Se puede observar también en el Algoritmo 6.10 que no se retorna el valor final de ninguna matriz, como acostumbra a pasar en los algoritmos de esta técnica. Bien, es lógico. En este caso toda la matriz entera se utiliza como componente de un cubo, o de una secuencia de matrices, de la qué la última iteración es la que nos proporciona el resultado, por tanto, en el fondo, ya es la rebanada final de una matriz tridimensional.

Por otra parte, si deseáramos obtener no sólo los caminos mínimos sinó también las aristas que los forman, sería preciso añadir una matriz de predecesores. Eso es, lo mismo que era un vector de predecesores para un árbol, pero una dimensión mayor. La matriz de predecesores tendría las mismas dimensiones que D, $n \times n$, igual que en el caso de Dijkstra. Nos daría los nodos de paso que hay en cada uno de los caminos entre dos vértices cualesquiera.

```
void floyd(grafo_pesos&g, matriz<double>& d)
{
    int n = g.tamano();
    matriz<double> D(1+n,1+n,oo);

    int u;
    para_todo_vertice(u,g) {
        para_todo_vecino(l,g[u]) {
            int v = l→v;
            double w = l→w;
            d[u][v] = w;
        }
        d[u][u] = 0;
    }

    for (int k=1; k<n; k++) {
        for (int u=1; u<=n; u++) {
            for (int v=1; v<=n; v++) {
                if (d[u][k] + d[k][v] < d[u][v]) d[u][v] = d[u][k] + d[k][v];
            }
        }
    }
}
```

Algoritmo 6.10 *Algoritmo de Floyd para los caminos mínimos.*

La eficiencia temporal del algoritmo de Floyd para los caminos mínimos entre cualquier par de vértices es $\Theta(n^3)$. La espacial, $\Omega(n^2)$.

Pesos Negativos

Como ya se ha dicho, el Algoritmo 6.10 funciona correctamente cuando en un grafo, por las razones que fueren, tengamos pesos negativos. Un comportamiento más crítico hay que tomar en el caso de encontrarse con grafos que contengan ciclos de peso negativo. Entonces el problema de los caminos mínimos no está definido para cualquier pareja de nodos que en alguno de los caminos que los unen se encuentre con algún ciclo de peso negativo.

El algoritmo de Floyd es capaz de detectar ciclos de peso negativo con un procedimiento $\Theta(n)$ adicional. Simplemente, después de cada iteración, o sea una vez obtenida cada D_k, hay que verificar que ningún elemento de la diagonal es negativo. Es decir, comprobar que $D_k[i,i] \geq 0$, $\forall i \in \{1,\ldots,n\}$. Si eso se cumple, entonces no hay problema. Si no, entonces el nodo i es incidente en algún ciclo de peso negativo, y el algoritmo se para.

En este capítulo se ha visto cómo llevar el principio de optimalitado al centro del razonamiento algorítmico. Se han mostrado problemas para los cuáles, dependiendo de los valores concretos de los datos, los algoritmos voraces pueden conseguir el óptimo o no. En cambio, con la programación dinámica podemos asegurar que se obtienen soluciones óptimas. Haciendo un uso intensivo de la memoria dinámica, la programación dinámica se provee de los óptimos de todos los subproblemas de la instancia inicial para tomar las decisiones correctas en una circunstancia dada. Estas decisiones se formulan como funciones de maximización o minimización de unas pocas opciones. También se ha puesto mucho énfasis a lo que se ha llamado naturaleza acumulativa de los planteamientos utilizados para resolver problemas con esta técnica.

Capítulo 7

Búsqueda Exhaustiva

En el Capítulo 5 se han visto los algoritmos voraces, un esquema algorítmico orientado a problemas de optimización en los cuáles cada una de las decisiones que se toma puede saberse óptima. Puede hacerse una analogía con cruzar un pasillo abriendo una serie de puertas, una tras otra. Después, en el Capítulo 6 se ha profundizado en la programación dinámica, esquema enfocado a problemas en los que en cada decisión se sabe que se toma la opción óptima, teniendo en cuenta los mejores resultados obtenidos en todas las decisiones anteriores. Es como si para subir una montaña, vamos siempre hacia el punto más alto que veamos. La búsqueda exhaustiva es útil cuando para saber que se está tomando una decisión óptima se necesitarían considerar también todas las decisiones futuras. Un laberinto.

La metodología de la búsqueda exhaustiva se compone de dos partes, *backtracking* y *branch and cut*. En español, los algoritmos de backtracking se conocen como algoritmos de vuelta atrás. Y a los algoritmos de branch and cut se les llama de ramificación y poda. Para referirnos a las dos estrategias en general se utiliza el concepto de búsqueda exhaustiva o también de algoritmos enumerativos. El hecho de que una cosa tenga muchos nombres es un claro indicador del uso que se hace de ella. Dicen que los esquimales tienen 250 maneras de decir blanco...

La característica principal de la vuelta atrás, es ser una técnica algorítmica para encontrar soluciones factibles, o decir que no existen, a problemas de decisión multietapa. Luego, no es una técnica para problemas de optimización, sinó de decisión. Esto convierte la técnica en más académica que aplicada. El quid de la cuestión es que hay que comprender el backtracking como primera etapa para poder aprender la técnica de ramificación y poda, que es la técnica aplicada más prolífica hoy día para problemas de optimización.

En la introducción de los problemas de optimización del Capítulo 5, se ha

señalado la utilidad de la completitud en funciones de variable continua. Se ha dicho que gracias a esta propiedad de los números reales podemos encontrar puntos singulares de funciones. Por contra, los valores que puede tomar una variable continua, no pueden ser enumerados. Y en el polo opuesto, podemos enumerar los valores que puede tomar una variable discreta. Gracias a la enumerabilidad de las variables discretas podemos encontrar puntos singulares de funciones. Y por contra, entre dos valores consecutivos de una variable discreta, no hay ningún otro valor posible, no hay completitud.

Pues bien, con la búsqueda exhaustiva, se nos ha acabado la cuerda. Ya no tenemos más imaginación y, exhaustos, recurrimos a la fuerza bruta. Haremos lo que haga falta para obtener el valor óptimo de las funciones. Aunque tarde siglos, trataremos todas las combinaciones posibles. Paciencia. Es una búsqueda exhaustiva, que agota.

Los modelos usados en estas metodologías son grafos implícitos, o infinitamente grandes. El grafo implícito que se utiliza sólo existe en la imaginación del programador. Materialmente, hay disponible la información relativa al nodo actual, y es calculable la manera de transitar a sus vecinos, o sucesores en el árbol de exploración. Todo ello recuerda de forma palmaria la inducción matemática.

Por tanto, las estructuras de datos diseñadas para representar grafos en el Capítulo 4, aquí no nos van a servir ni poco ni mucho. Sólo para problemas chiquititos. Para los grandes, no sabemos a priori cuántos nodos tendrá el grafo que nos disponemos a explorar ni cuántos vecinos cada nodo... Tan solo conocemos un nodo inicial, y la manera de pasar de un nodo a otro, o sea, la manera de construir aristas del grafo. Para explorarlo, es preciso establecer una ordenación entre los vecinos de cada nodo. Visitarlos en algún orden prestablecido evita repeticiones. Si el grafo tiene ciclos, el programa se cuelga. Entonces se dice que el programa *ciclea*. Para impedirlo, hay diferentes mecanismos. Que la exploración describa un grafo dirigido ya ayuda bastante. Además, para detectar un ciclo de longitud superior a dos arcos, se requieren estructuras de datos específicas.

El capítulo abre con una sección de preliminares. Como tales, se hace un repaso del protocolo que sigue un sistema operativo, en colaboración con la arquitectura del ordenador, cada vez que una aplicación hace una llamada a una subrutina. Y también se esboza una idea inicial para hacer un programa que juegue al ajedrez. Este par de referencias nos permitirán aterrizar en el mundo de la búsqueda exhaustiva adecuadamente.

Después, el capítulo se divide en dos partes. En la primera se estudia la vuelta atrás. Se propone un esquema algorítmico para el backtracking y se apuntan características comunes a los algoritmos que se ajustan a esta estrategia. Hay entonces un análisis minucioso del problema de las ocho reinas y se muestra un algoritmo con una primera solución, iterativa y muy grosera, por su simplicidad. Rápidamente, se generaliza el problema de las ocho reinas al problema de las n reinas, en lugar de ocho en particular. El backtracking concluye

con el problema del laberinto. La táctica utilizada en la implementación de este problema casa perfectamente con el esquema algorítmico de vuelta atrás. Es un buen paradigma, y evidencía el alcance de esta metodología.

Llegados a este punto, enriquecemos el backtracking adentrándonos en el branch and cut. Al finalizar el capítulo nos daremos cuenta de que el esquema de vuelta atrás es tan solo la introducción a la ramificación y poda, ya que forma parte de ella. La ramificación y poda, es sin duda lo que más debería trascender de todo el libro. Si de aquí un par de años aún recordáis alguna cosa, la mejor de todo el libro sería ésta.

Se denomina ramificación y poda en referencia al árbol de exploración, insinuando que estos algoritmos no hacen más que irse por las ramas. Se habla de relajaciones, se introduce el esquema algorítmico, y tres ejemplos de sus posibilidades. El último algoritmo que se ve en el capítulo es una implementación en la metodología de ramificación y poda, para el súper famoso problema del viajante, en inglés *the Traveling Salesman Problem*, o directamente TSP. El capítulo se despide con unas nociones breves de programación matemática, introduciendo el concepto de modelo poliédrico de un problema de optimización.

7.1 Preliminares

En esta introducción se muestra la estrecha relación que hay entre las implementaciones recursivas y los problemas de búsqueda exhaustiva. Consecuentemente, a lo largo de todo el capítulo se tratan tanto la vuelta atrás como la ramificación y poda recursivamente. No obstante, es posible implementar algoritmos para solucionar los mismos problemas con métodos iterativos. Para eso, son necesarias estructuras sofisticadas como pilas o colas de prioridad.

7.1.1 Comportamiento Recursivo

Antes de introducirnos en los contenidos troncales del capítulo, conviene aquí hacer un repaso del protocolo respeto al paso de parámetros que se sigue cuando hay llamadas entre rutinas de un programa informático. Es bien sabido que hay dos formas de pasar parámetros, por valor y por referencia.

Pasar parámetros por referencia en aplicaciones cerradas es superfluo. Cualquier parámetro pasado por referencia puede ser sustituido por una variable global. Hay quien considera, con razón, que las variables globales ensucian la modularidad. Pero bien, por eso se inventó la programación orientada a objetos. Las clases introducen un nivel intermedio de visibilidad. Una variable miembro en una clase es una variable global en el ámbito de la clase. A lo largo de este capítulo utilizaremos clases y variables miembro, y evitaremos pasar parámetros

por referencia.

Prestemos atención al aspecto que puede tener una llamada cualquiera a una rutina desde un módulo principal. Por ejemplo $factorial(k)$. Y nos ponemos una cuestión, ¿Podemos saber, a partir de la información de esta llamada, si el paso del parámetro k es por valor o por referencia?. La respuesta es, No. No podemos saberlo, ya que tan solo con esta línea de código, no sabemos si la k es una constante o una variable.

- Si supiésemos seguro que k es una constante, entonces, podríamos asegurar que la rutina recibe el parámetro por valor.
- Si supiésemos seguro que k es una variable, entonces no podríamos saber si el parámetro se está pasando por valor o por referencia.

Ahora bien, si en el parámetro de la llamada aparecen operadores matemáticos, entonces canta. La llamada $factorial(k+n)$ invoca una rutina pasándole el parámetro por valor. No cabe la menor duda.

Hay un comportamiento específico de los algoritmos recursivos que hay que tener muy claro para comprender la búsqueda exhaustiva. Este comportamiento se explica en el ejemplo siguiente.

```
void r(int i)
{
     imprimir(i);
     if (i<5) r(i+1);
     imprimir(i);
}
```

Algoritmo 7.1 *Comportamiento recursivo.* 1 2 3 4 5 5 4 3 2 1 .

Un programa que llamase a la rutina del Algoritmo 7.1 pasándole un 1, o sea, que llamara a $r(1)$, obtendría la salida 1 2 3 4 5 5 4 3 2 1 . Es interesante. La secuencia de salida sube y baja obedeciendo una simetría que en el código no se manifesta, ya que hay un signo +, pero ningún signo -.

En la función del Algoritmo 7.1 sabemos que el paso de parámetros es por valor por dos razones. Una, evidente, porqué tenemos la cabecera ante los ojos. La única manera de pasar parámetros por referencia que sean de tipos primitivos como char, short, int, double, etcétera, es añadiendo un & que en el parámetro formal i de la cabecera no aparece. Si la cabecera fuese *void r(int& i)*, entonces efectivamente sí que sería un paso de parámetro por referencia, y la salida ya no sería la misma, sinó 1 2 3 4 5 5 5 5 5 5 .

La primera razón, pues, es la evidente. La segunda es deducible. Ya que en el parámetro actual de la llamada aparece una operación matemática, $i + 1$, el valor de la variable i jamás puede haber sido modificado cuando se vuelva de esta rutina recursiva. Por tanto la salida es 1 2 3 4 5 5 4 3 2 1 , como debe ser.

Conviene hacerse una imagen mental de cómo funciona la recursividad para asumir interiormente este fenómeno. En la Figura 7.1 hay un esquema del flujo de ejecución de la llamada $r(1)$. Es interesante recordar la recursividad así, desplegada, para analizar procesos.

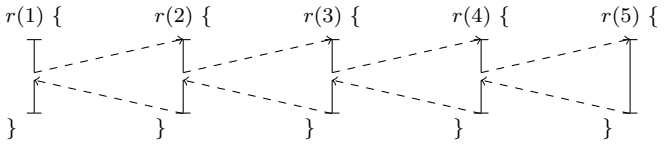


Figura 7.1: *Desplegamiento del flujo de una llamada recursiva.*

Arquitectura

Para profundizar en el por qué de todo ello recurrimos a la arquitectura de computadores.

Cuando un programa realiza una llamada a una rutina se crea en la pila del sistema operativo una estructura denominada *bloque de activación* de la rutina. Como en cualquier pila, las cosas se quitan en el orden inverso a como se hayan puesto. En esa inversión reside la esencia de la vuelta atrás.

Los elementos que se ponen en la pila son bloques de activación. Se añade un bloque en cada llamada, apilándose los bloques tal como se anidan las llamadas. Así pues, mientras corre el programa principal, *main()*, la pila está vacía. Cuando se llama una rutina, se añade a la pila un elemento de tamaño impredecible. Depende de los parámetros que se le pasen a la rutina y de sus variables locales. En cualquier caso, si la primera rutina que se ha llamado hace una llamada a una segunda rutina, la pila pasará a tener dos bloques de activación. Y ya no dejará de tener dos o más, hasta que no se retorne de la primera llamada al programa principal.

Por tanto, el número de bloques de activación que hay en la pila refleja en todo momento el nivel de anidamiento de rutinas en el proceso. Es fácil saber ese nivel a partir del encadenamiento de los apuntadores a la pila, *stack pointer*, en inglés, o directamente SP. El apuntador a la pila nos indica en todo momento donde empiezan nuestras variables locales. Y por lo tanto, cuál es el espacio de memoria que desaparecerá, bien, que se liberará, cuando se retorne del procedimiento que estamos ejecutando. Conclusión, cuando retornamos de una función o procedimiento, las variables locales y los parámetros dejan de existir.

Hay que entender que desde dentro de la rutina, para ella, los parámetros por valor no son más variables locales inicializadas desde fuera. Si después de todo queda alguna consecuencia en alguna variable del módulo que llama a la rutina es porqué se ha pasado el parámetro por referencia, y en el bloque de activación de la pila ha estado habiendo la dirección de la variable modificada.

Involucrar en todo este protocolo el apuntador a la pila del sistema, representa introducir aspectos de la arquitectura física del procesador, del cableado, ya que como es sabido, este valor, que al fin y al cabo contiene una dirección de memoria, se guarda en el corazón del sistema, dentro de la CPU. Es en este punto donde se hace visible la colaboración de la arquitectura del computador.

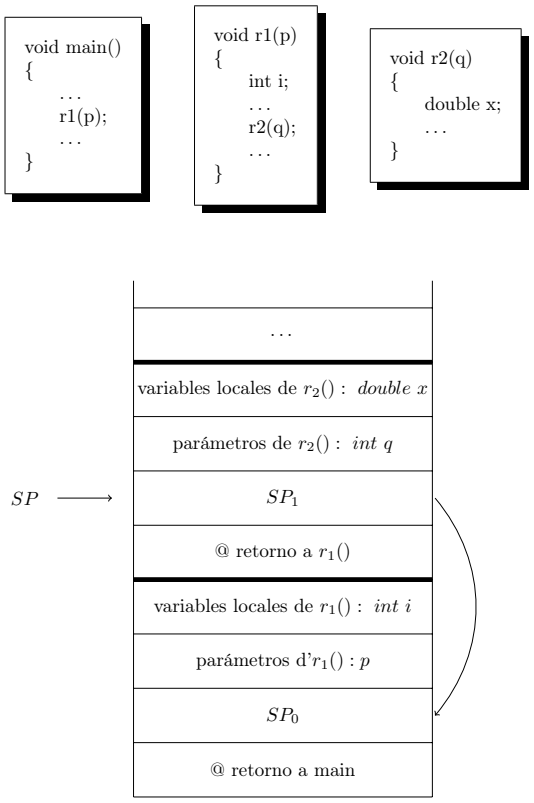


Figura 7.2: *Bloques de activación anidados en la pila del sistema operativo.*

En la Figura 7.2 se puede ver un esquema del funcionamiento. El estado de la pila indica que se está ejecutando $r_2()$. Se utiliza trazo grueso para separar los bloques de activación. Hay dos, uno para cada rutina.

Una estructura de pila se representa de abajo arriba. La estructura de un bloque contiene en primer lugar, o sea debajo, la dirección de retorno para poder seguir el flujo de ejecución cuando se retorne de la rutina. Después, el valor que tenía el apuntador a la pila en la llamada anterior. Eso sirve para encadenar los bloques. Después, se guardan primero los parámetros y

encima las variables locales de la rutina llamada. Todo ello tiene consecuencias, especialmente para los compiladores. Ellos son quienes traducen cada tipo de variable en una cantidad fija de bytes.

En síntesis, parámetros por valor son variables locales sólo accesibles desde dentro de la rutina mientras se esté ejecutando. Y por tanto, cuando se pasan parámetros recursivos por valor se produce un fenómeno de vuelta atrás, 1 2 3 4 5 5 4 3 2 1 . Cada uno de estos números ha sido guardado en un bloque de activación distinto, llegándose a apilar los cinco bloques en el momento central.

7.1.2 Jugar al Ajedrez

¿Cuántas jugadas posibles hay como primera jugada en una partida de ajedrez?

Bien, comienzan las blancas. Tienen 8 peones, cada uno de los cuáles tiene inicialmente dos posiciones posibles. Son 16. Además, el jugador con blancas puede comenzar moviendo cualquiera de los dos caballos. Cada caballo tiene también dos posiciones posibles. En total son 20.

Y ¿De cuántas maneras puede responder el jugador con negras? Con las 20 correspondientes a sus fichas, claro.

¿Cuántas posibles partidas distintas pueden existir después de que las negras hayan tirado por primera vez? Pues 400, 20 por 20.

Y ¿Cuántas jugadas diferentes pueden hacer las blancas entonces?

Eso ya no se puede decir fácilmente. Depende de las dos primeras jugadas.

Para poderlo responder, imaginemos un grafo implícito como el de la Figura 7.3.

Se trata de un grafo dirigido donde cada nodo representa una disposición concreta de las piezas en el tablero, y cada arco un cierto movimiento de alguna pieza. El nodo central corresponde a la posición inicial de cualquier partida. Viene dado por el reglamento del juego. Por lo que se ha comentado, del nodo central de la figura salen 20 vecinos, o sucesores en el árbol de exploración.

A primera vista ya se observa que el grafo del modelo es implícito y dirigido. En un análisis un poco más minucioso, en seguida vemos que de los cuatro nodos correspondientes a jugadas donde se han movido los caballos, en la Figura 7.3 los cuatro nodos superiores del ala izquierda, hay arcos que retornan a la jugada inicial, provocando así ciclos. O sea, que además de un grafo implícito y dirigido, nos enfrentamos a un grafo con ciclos.

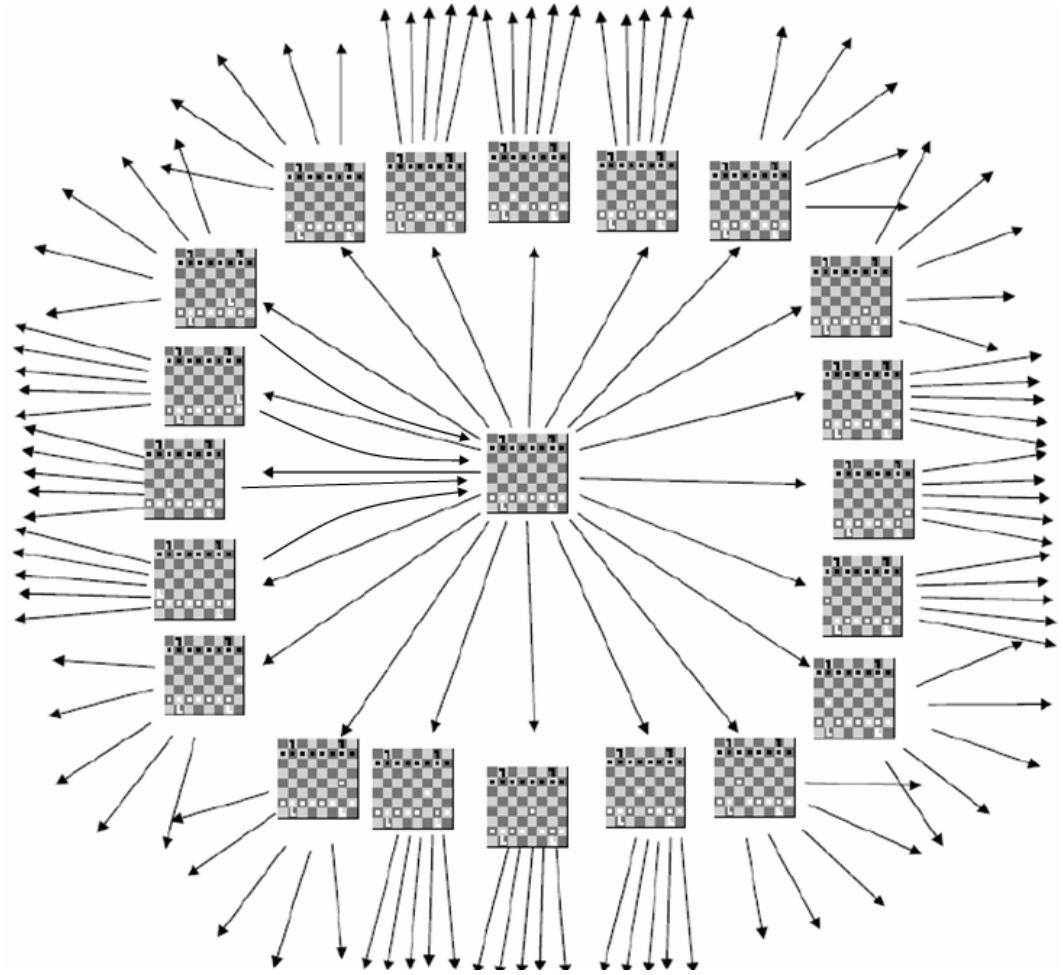


Figura 7.3: *Grafo implícito que representa todas las partidas de ajedrez posibles.*

No se trata de profundizar demasiado en este desarrollo. Es tan solo un esbozo de la naturaleza de los planteamientos que se atacan con estrategias enumerativas.

Este modelo es un paradigma que ilustra la filosofía de los algoritmos de vuelta atrás. También sirve para hacernos una idea de la cantidad colosal de tiempo que pueden requerir esos algoritmos si no hacemos algo para evitarlo.

7.2 Vuelta Atrás

El esquema algorítmico de vuelta atrás sirve para elaborar algoritmos que resuelven problemas de decisión. Eso lo hacen utilizando grafos implícitos como

modelos. En implementaciones recursivas, cada anidamiento de la llamada recursiva significa un nodo, y los parámetros entre llamadas representan aristas.

Para problemas de decisión con variables binarias exclusivamente, y no enteras en general, el grafo que se modela es un árbol binario. En cada paso tan solo hay que visitar dos nodos vecinos, el correspondiente a tomar la siguiente decisión afirmativamente, $x_{i+1} = \mathit{cierto}$, y el de rechazarla, $x_{i+1} = \mathit{falso}$. En ese caso se puede evitar el bucle de los casos recursivos.

Por otro lado, y eso va más allá que la búsqueda exhaustiva, en cualquier metodología para problemas de optimización se puede transformar cualquier problema de variables enteras en uno de variables binarias, siempre y cuando tengamos alguna cota superior para las variables. Por ejemplo, si tenemos una variable x_1 que puede tomar valores enteros entre 1 y 7, entonces la podemos transformar en 7 variables binarias y_j, para $j = 1, \ldots, 7$, cada una de las cuáles puede valer 0 o 1 (claro, si son binarias...), y posteriormente a la exploración, hacer la suma de las 7 variables para saber el valor de la variable inicial x_1. O sea, que teóricamente podríamos resolver cualquier problema con un algoritmo de vuelta atrás en el que cada nodo sólo tuviera dos sucesores, siempre que se disponga de cotas superiores para las variables iniciales. Teóricamente.

Clásicamente se distinge entre dos tipos de problemas para los cuales es útil el esquema de backtracking.

- *Problemas de solución única.* Aquellos que comprueban si el problema planteado tiene alguna solución, o no.
- *Problemas de múltiples soluciones.* Aquellos que buscan todas las soluciones posibles.

Lógicamente, no hay ninguna diferencia entre los dos tipos de problemas si resulta que el problema no tiene ninguna solución. O, dicho de otra forma, la eficiencia en el peor de los casos para los dos tipos de algoritmos es la misma.

7.2.1 Esquema Algorítmico de Vuelta Atrás

En su forma más genérica, el esquema algorítmico de la búsqueda exhaustiva es casi clavado a un DFS. Y sin el casi. Conviene recordar que básicamente

una búsqueda exhaustiva es un DFS de un grafo implícito.

De hecho, ni tan solo es necesario que el grafo sea implícito. La búsqueda exhaustiva es un DFS. Si el grafo cabe en memoria, no hace falta decir nada más. Ya se ha visto como se puede recorrer en el Capítulo 4. Cuando para resolver el problema hay que modelar un grafo implícito, es cuando la búsqueda exhaustiva añade nuevo conocimiento.

En el Esquema 7.1 se muestra una propuesta recursiva para algoritmos de vuelta atrás.

```
algoritmo backtracking(i)
{
    si (i = n) {
        X ← X^i
    }
    sino {
        para v = cada valor posible de x_{i+1} {
            x_{i+1} ← v
            si (factible(X^i ∪ {x_{i+1}})) {
                X^{i+1} ← X^i ∪ {x_{i+1}}
                backtracking(i + 1)
            }
        }
    }
}
```

Esquema 7.1 *vuelta atrás.*

Suponiendo que se tiene una solución factible desde 1 hasta i, los algoritmos de búsqueda exhaustiva se dedican a encontrar una de factible hasta $i + 1$. Razonamiento inductivo por antonomasia.

Fisiológicamente, estos algoritmos acostumbran a ceñirse al esquema más que en cualquier otra técnica. Se acostumbra a poner, como aquí, el caso trivial al inicio. Casi siempre es el mismo, cuando el parámetro i valga n.

Los actores que participan en este escenario se enumeran seguidamente.

- El parámetro por valor i: Significa cuántas decisiones han sido tomadas en el nodo actual, y coincide con el tamaño del subproblema resuelto hasta ahora.
- Variable global n: Es el tamaño de la instancia, el número de decisiones que hay que tomar.
- Vector solución X: Tiene dimensión n. Guarda la solución en el momento en que se obtiene. Sólo aparece en el caso trivial.
- Vector subsolución actual X^i: Tiene dimensión i. Guarda la subsolución factible actual. Contiene el valor de las i decisiones ya tomadas. Representa el nodo actual en el grafo de exploración.

- Variable de decisión x_{i+1}: Es la nueva decisión que se toma en la llamada actual. Cada valor posible v de x_{i+1} representa un nodo vecino del nodo actual. Para cada posible valor que pueda tomar abriremos una nueva exploración.

La función *factible(i)* viene determinada por el problema que se pretende resolver.

Los algoritmos de backtracking arrancan a partir de un módulo principal con la llamada *backtracking(0)*, y el vector X^0 vacío. Por esta razón conviene utilizar clases y agrupar todas esas variables. El constructor de la clase recibirá los datos de la instancia y será quién haga la llamada inicial. Todos los vectores X^i que se utilicen en la exploración compartirán el mismo espacio. Eso es un solo vector que, como la rutina con el backtracking, serán privados. La solución X será pública. El funcionamiento canónico pues, será crear un objeto de la clase, y pedirle los resultados.

Si solamente se buscase una solución, habría que añadir una variable global *encontrada* en el Esquema 7.1. El módulo principal la inicializaría a *falso*, en el caso trivial se pondría a cierto, y condicionaría la misma alternativa donde se testea la factibilidad con una conjuntiva AND.

Si se buscasen todas las soluciones posibles, deberíamos imprimirlas en lugar de salvarlas en el vector X, para que no se machacasen.

Marcajes

Cuando además de la subsolución X^i en cada nodo se utiliza información adicional para resolver la factibilidad, entonces se puede utilizar otras variables globales, que seguramente serán vectores. Pero así como cualquier combinación de valores en el vector X^i respresenta alguna solución, el valor de estos vectores puede requerir mantener una integridad que el comportamiento recursivo no le garantiza. En estos casos, se añade código simétrico antes y después de la llamada recursiva del Esquema 7.1.

Eficiencia

El inconveniente más importante de esta metodología es el tiempo. Ya se ve que esto tardará mucho. Para el caso de variables binarias, $\Omega(2^n)$. Puede ser calculado con el Teorema Maestro de las recurrencias substratores, con $a = 2$, $c = 1$, y $k = 1$. Esto es, como $a > 1$, vamos por el caso $\Theta(a^{n/c})$. Esta desventaja se agrava si además, usamos variables enteras como sumas de variables binarias tal como se ha mencionado más arriba. No es nada satisfactorio analizar la eficiencia de los algoritmos de búsqueda exhaustiva. Por ejemplo, la implementación

para el problema de las n reinas que se presenta finalmente en el Algoritmo 7.3 es $\Omega(n!)$.

Finalmente hemos caído de cuatro patas en el centro de aquello que a lo largo de los capítulos anteriores hemos intentado evitar, tardar un tiempo que no se puede acotar polinómicamente con el tamaño de la entrada. Llegados a este punto, la única cosa que nos preocupará es aproximarnos a las eficiencias polinómicas, y aun así, no lo conseguiremos. No es de extrañar que en adelante, nos olvidemos de los análisis de las eficiencias. Ya hemos perdido la batalla, y tan solo nos queda hacer lo que seamos capaces para salir tan airosos como sea posible.

7.2.2 El Problema de las Ocho Reinas

Como en el capítulo de programación dinámica, comenzamos ilustrando la técnica algorítmica mediante un problema que se soluciona mirándose al espejo del esquema algorítmico.

Definición 7.1 El Problema de las Ocho Reinas. *Colocar ocho reinas en un tablero de ajedrez sin que ninguna amenace a otra.*

Ahora bien, según lo que en este libro se considera problema, el problema de las ocho reinas como tal es un caso patológico, ya que no tiene ninguna entrada de datos. Cuando no hay datos de entrada, $n = 0$, y entonces podemos analizar la eficiencia del algoritmo sin ni siquiera mirarlo. La eficiencia es $\Theta(1)$. Todos los algoritmos del mundo que no tienen datos de entrada, tienen una eficiencia de $\Theta(1)$.

Para traerlo a nuestro territorio conviene generalizar el problema. En lugar de tener un tablero de ocho por ocho casillas y tener que colocar ocho reinas, consideraremos tableros de n por n, y el problema será emplazar n reinas. No obstante, si en el título de esta sección dice ocho, y no n, es porque para el caso concreto de ocho reinas, o de cualquier número fijo, también se hacen algunas reflexiones que conviene tener en cuenta. Reflexiones que iluminan el por qué de la búsqueda exhaustiva. Más adelante en esta sección, se da una implementación del problema de las n reinas. Pero bien, empecemos por el principio.

Hay una manera, la peor, de resolver este problema. Es una lástima que sea la peor, ya que resulta de una legibilidad exquisita. Es un caso parecido a la resolución de Fibonacci con una función recursiva como la de la página 133, que el aspecto fisiológico del código es de lo más deseable, pero la ineficiencia lo hace inadmisible.

La manera más grosera de resolver este problema es poniendo ocho bucles anidados. Un código como

```
bool ocho_reinas()
{
    for (int i=1; i<=8; i++)
        for (int j=1; j<=8; j++)
            for (int k=1; k<=8; k++)
                for (int l=1; l<=8; l++)
                    for (int m=1; m<=8; m++)
                        for (int n=1; n<=8; n++)
                            for (int o=1; o<=8; o++)
                                for (int p=1; p<=8; p++)
                                    if (solucion(i,j,k,l,m,n,o,p) return true;
    return false;
}
```

hace que el problema sea fácil. La función *solucion()* debería mirar que para cada pareja de las[1] 28 posibles entre las ocho reinas, ninguna amenace a otra. Sin duda, el problema quedaría solucionado.

Desde una perspectiva más filosófica, resulta muy interesante observar que si en lugar de ocho fuesen n reinas, este planteamiento ya no nos serviría, ya que el número de bucles anidados dependería de n. Y claro, es imposible hacer un programa con un número variable de bucles anidados. Esto es importante. Esta imposibilidad podría ser sorteada haciendo un programa que sirviese tan solo para escribir códigos de programas... pero algo huele a retorcido, y se intuye que es mejor adandonar esta aproximación. No obstante, de esta disertación conviene retener la idea de que la búsqueda exhaustiva es útil para cuando se requieren un número variable de bucles anidados.

Empecemos de nuevo el análisis del problema desde otra óptica. Observemos rápidamente que las ocho reinas deberán estar en filas distintas. O sea, que se puede representar una solución en un vector de ocho posiciones, $T[i]$, para $i = 1, \ldots, 8$. En $T[i]$ se guarda la columna de la reina de la fila i en el tablero. Para hacerlo más compacto todavía, se podrían representar las soluciones en un solo número de ocho cifras. Tendría que ser un número que no tuviera ningún nueve ni ningún cero. Además, todas las cifras deberían ser distintas. Así, se puede resolver el problema con un algoritmo iterativo. O sea, solucionar este problema es muy fácil, sólo es necesario un contador que cuente hasta 88888888 y para cada valor, mirar si las posiciones de reinas que representa, es factible. Para aligerar la tarea, en lugar de empezar a contar por el 1, se podría comenzar directamente por el 12345678, que es el número menor que satisface las restricciones de filas y columnas, aunque no las relativas a las diagonales. Igualmente, se podría acabar en el 87654321, ya que si entonces no hemos encontrado ninguna solución, es que no la hay.

Es seguro que dos reinas no estarán en la misma fila, ya que cada una corresponde a un índice distinto del vector solución. Dos reinas estarán en la misma columna si $T[i] = T[j]$. Quedan por identificar las amenazas en

[1] 8*7 / 2*1

diagonal. Del análisis matemático, sabemos que la función $f(x) = x$ es una diagonal del plano cartesiano, y $f(x) = -x$, la otra. Y claro, $f(x) = x+1$ es una diagonal una poco más arriba. Las rectas $f(x) = x + K$ o $f(x) = -x + K$, para alguna constante K, son paralelas a las diagonales. Es decir, a las bisectrices de los cuadrantes.

En definitiva, a partir del vector T, y de las posiciones de reinas que representa, podemos detectar que dos reinas se amenazan en diagonal añadiendo en el código un par de sentencias alternativas.

Dos reinas en las filas i y j, y columnas $T[i]$ y $T[j]$ del tablero, se amenazan en diagonal si

- $T[i] + i == T[j] + j$, por la diagonal principal ($\searrow$).
- $T[i] - i == T[j] - j$, por la diagonal secundaria ($\nearrow$).

En la clase implementada en el Algoritmo 7.2 hay el código correspondiente a esta segunda propuesta. Es también muy basta, como la anterior, la de los ocho bucles anidados mencionada antes. El interés yace en la enumerabilidad. Dejar claro por qué a los algoritmos de búsqueda exhaustiva también se les conoce como algoritmos enumerativos.

El programa para resolver el problema se limitaría a crear un objeto de la clase del Algoritmo 7.2, y pedir por el valor de la solución. O sea, todo el trabajo se hace en el constructor. Hay tan solo un contador que recorre todo el espacio que se ha visto. Es decir, desde 12345678 hasta 87654321 como máximo, si no existiese solución.

En la función miembro *factible()* del Algoritmo 7.2 nos aprovechamos de que un número natural, al crecer, pase por todas las ordenaciones posibles de sus cifras.

El tipo *unsigned integer* quizás no sea necesario, pero refuerza la legibilidad. A ver. Hagamos números. Un entero se guarda en cuatro bytes, 32 bits. En otras palabras, puede representar los valores positivos, o sea sin signo, del 0, al $2^{32} - 1$. Son 32 cifras binarias. Como ya se ha dicho, eso son entre 10 y 11 cifras decimales, ya que el $log_2(10)$ es 3 y pico, y 32 dividido por 3 y pico es 10 y poco. Para el caso que nos ocupa, hay que trabajar con números de ocho cifras máximo. Si utilizamos un entero tal cual, *int*, entonces tenemos la mitad de representación que sin signo, o sea, desde -2^{31} hasta $2^{31} - 1$. Total, que efectivamente en el Algoritmo 7.2 el parámetro se hubiese podido declarar de tipo entero normal, o sea con signo. Pero tal como se ha dicho, así refuerza la legibilidad, y serviría para n's más grandes que ocho. Poco más grandes, 9 y tal vez 10.

```
class ocho_reinas {

    bool factible(unsigned int s){
        int T[8];
        int i = s;
        for (int j=0; j<8; j++) {
            T[j] = i % 10;
            i = i / 10;
            if (T[j] == 0 || T[j] == 9) return false;
            for (int k=0; k<j; k++)
                if  (T[k] == T[j] ||
                    T[k]-k == T[j]-j ||
                    T[k]+k == T[j]+j) return false;
        }
        return true;
    }

public:
    bool encontrada;
    unsigned int z;
    ocho_reinas() {
        z=12345678;
        while (!factible(z++) && z<87654321);
        z--;
        encontrada = (z<87654321);
    }
};
```

Algoritmo 7.2 *Algoritmo enumerativo para el problema de las ocho reinas.*

Por otra parte, está bien claro que cuando se encuentra una solución se están encontrando cuatro tan solo girando el tablero, por simetría. Seguramente esto permitiría reducir todavía más el ámbito de búsqueda.

Sin embargo, no olvidemos que estas dos primeras ideas que se dan para el problema de las ocho reinas no son las que finalmente nos quedaremos. En definitiva, se están citando alternativas para resolver un mismo problema. Y todas ellas participan del esquema enumerativo.

En la Figura 7.4 se muestra la sexta solución, por orden ascendente, del problema, 25713864. Si se ejecuta el código que viene con este libro, se puede obtener soluciones para cualquier n, que quepa.

Con todo el rigor, la eficiencia del constructor de la clase del Algoritmo 7.2 es $\Theta(1)$, ya que el número de reinas es constante, y por tanto, todo ello.

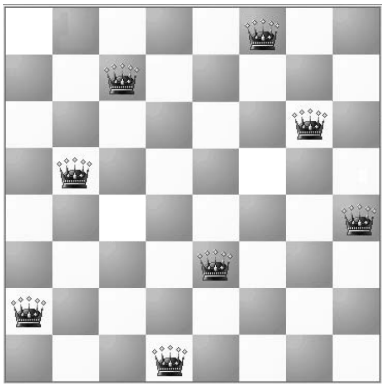


Figura 7.4: *Solución* 25713864 *del problema de las ocho reinas.*

El Problema de las n Reinas

Se presenta seguidamente una generalización del enunciado del problema anterior en la que se pretende establecer distancia con el juego del ajedrez.

Definición 7.2 El Problema de las n Reinas. *Marcar n casillas de una matriz de $n \times n$ de manera que tanto en cada fila, como en cada columna, como en cada una de las dos diagonales módulo n, tan solo haya una marca.*

Echemos un vistazo al primer problema resuelto siguiendo el esquema de vuelta atrás al pie de la letra, en la clase *n_ reinas*.

En el Algoritmo 7.3 se resuelve el problema de las n reinas. Esta clase escribe por pantalla una lista enumerada de soluciones. Para la tarea de escribir utiliza una función externa a la que se le pasan dos parámetros, *imprimir(vector<int> T, int ns)*. El primero caracteriza la solución propiamente dicha, y el segundo, *ns*, es para mantener el contador de soluciones por pantalla. Tal como está en el código que se subministra con este libro, en el fichero *7.busqueda_ exhaustiva.cpp* se encuentra esta función, en el programa principal del capítulo 7.

La clase comienza declarando una variable miembro privada, un contador de soluciones encontradas, *ns*. Funcionalmente, tan solo sirve para la impresión de las soluciones, aunque luego también se utiliza en la función booleana *hay_ solucion()*. Lo importante es la variable miembro T, vector de enteros, que es el vector de subsoluciones parciales X^i del Esquema 7.1. Un nodo del árbol de exploración viene caracterizado por su nivel, i, y por su solución parcial, $T[1..i]$. Este vector de soluciones parciales se declara público para evitar envoltorios.

Además, la clase tiene tres funciones miembro. El constructor y dos funciones más. Disponer de la función externa para imprimir soluciones nos libera de tener

que utilizar otras variables para salvar las soluciones finales.

- La primera de las funciones miembro, *factible()*, tiene la misión de verificar la factibilidad de una subsecuencia de decisiones de longitud i.
- La segunda, *busca()*, hace el backtracking.
- El constructor arranca el backtracking con $i = 0$, es decir, a partir de una subsecuencia inicial vacía, $X^0 = \emptyset$.

```
extern void imprimir(vector<int> T, int ns);

clas n_reinas {
    int ns;

    bool factible(int i) {
        for (int j=0; j<i; j++)
            if (T[i]==T[j] || T[i]-i==T[j]-j || T[i]+i==T[j]+j) return false;
        return true;
    }

    void busca(int i) {
        if (i==n) {
            imprimir(T,++ns);
            return;
        }
        for (int j=0; j<n; ++j) {
            T[i] = j;
            if (factible(i)) busca(i+1);
        }
    }

public:
    vector<int> T;

    n_reinas(int n) {
        ns = 0;
        T.crea(n);
        busca(0);
    }
    ~n_reinas() { T.destruye(); }
    bool hay_solucion() { return (ns>0); }
};
```

Algoritmo 7.3 *Algoritmo enumerativo para el problema de las n reinas.*

Referente exclusivamente a la función *busca()*, observad que se toma una decisión en cada fila. Tenemos que n es el número de decisiones que hay que tomar, el tamaño de la instancia. En cada nodo del árbol de exploración habrá tantas decisiones tomadas, casillas marcadas, como el nivel del nodo en el árbol, o de anidamiento en la rutina recursiva. Estas soluciones parciales se guardan, para cada nueva decisión i, en las posiciones $T[1..i]$ del vector. Así pues, la solución quedará en el vector T de una manera análoga a como ya se ha hecho para el problema concreto con 8 reinas. No obstante, antes el vector T tenía las 8 componentes llenas en todo momento, conteniendo también soluciones no factibles. Ahora no. Aquí el subproblema resuelto es factible en todo momento, y es la cantidad de componentes llenas en el del vector lo que irá creciendo. Por otra parte, hay que diferenciar. Una cosa es el tamaño de la instancia, las n decisiones, y otra muy distinta los n valores posibles para cada decisión. En otras palabras, además de que n sea la profundidad del árbol de exploración, también es un árbol n-ario. Entre un nodo, y cada uno de sus n sucesores la casilla adicional marcada estará en una columna diferente de las n posibles.

Esta implementación del algoritmo para el problema de las n reinas es elegante, breve y reglada. Aún así, no es una solución polinómica. Aunque dé vergüenza, se está presentando un algoritmo $\Omega(n^n)$. Más lento, imposible. A saber cuánto tardaría para $n = 100$!

7.2.3 El Laberinto

No se puede conocer el futuro, pero sí que se puede suponer que suceden cada una de las opciones posibles, y prepararse para actuar en consecuencia.

Definición 7.3 Problema del Laberinto. *Dada una matriz L de $m \times n$ valores $\{0, 1\}$ que indican que se puede pasar por la casilla (i, j), cuando $L(i, j) = 1$, o no, $L(i, j) = 0$, para $i = 1, \ldots, m$ y $j = 1, \ldots, n$, averiguar si hay algún camino para ir de la casilla $(1, 1)$ a la (m, n). Un camino es una secuencia de casillas vecinas por las que se pueda pasar. Vecinas por filas o columnas, no por diagonales.*

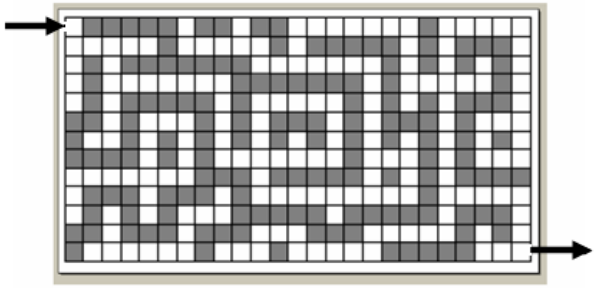


Figura 7.5: *Laberinto.*

El método que se utiliza aquí para resolver el problema del laberinto pasa por representar el laberinto en un grafo con listas de adyacencia, Sección 4.2.2. El nuevo grafo contiene un vértice para cada casilla de la matriz, $m*n$ vértices. Una arista entre dos vértices significa que las casillas correspondientes son vecinas. Las casillas prohibidas resultarán vértices aislados. Entonces, el problema se resolverá haciendo un recorrido en profundidad del grafo. Una casilla (i, j) del laberinto se corresponde con el nodo v en el nuevo grafo cuando $v = i * n + j$. O sea, un nodo v del grafo, representa la casilla $(v/n, v\%n)$.

En el Algoritmo 7.4 hay implementada la clase *laberinto*. Las variables miembro, todas privadas, son las que se comentan seguidamente.

- La variable booleana *encontrada*, ya que se trata de un problema de solución única. Como siempre, esta variable abortará la exploración en el momento en que se encuentre la primera solución factible.
- Las medidas m y n, del laberinto. Se inicializan en la creación de los objetos de la clase laberinto, y no se tocan más.
- Una matriz de enteros T. Es la matriz de entrada de dimensiones $m \times n$, pasada al constructor por referencia, como todas las matrices. La matriz apuntada por T viene a ser un parámetro de entrada y salida de la clase. Además de para crear el grafo (parámetro de entrada), también se utiliza para rellenar las casillas de la solución con el valor especial -1 (parámetro de salida).
- El vector x con las soluciones parciales donde finalmente quedará la solución en forma de vector de predecesores.
- El vector de listas *grafo* representa el mismo laberinto de entrada. La estructura *lista* a la qual se hace referencia, se puede consultar en la página 64, aunque para seguir el hilo, no es preciso.
- En la función que realiza el backtracking, *busca()*, se utiliza un vector de booleanos, c, que para cada casilla nos indica si ya ha sido visitada, o no.

El constructor recibe la matriz con el laberinto a partir de la dirección $\_T$. Hace la inicialización de la variable *encontrada* y toma nota del valor de los parámetros. Seguidamente crea el grafo con la función *crear_grafo()*, inicializa los vectores, y llama a *busca(0)*. Cuando ha obtenido el vector de predecesores x, modifica la matriz de entrada poniendo menos unos en el camino solución. Y, como detalle de calidad, deja el número de pasos que tiene el camino en una variable pública l. En la función miembro auxiliar para crear el grafo, un par de bucles anidados recorren todas las casillas de la matriz de entrada, es decir, todos los vértices del grafo. A cada uno de ellos se le inicializa su lista de adyacencia, y se le añaden tantas aristas como casillas vecinas tenga la casilla asociada a ese vértice, con la función miembro *pon()* de la estructura lista. Para las casillas prohibidas, $T[i][j]$ es falso, y la lista de estos nodos, vacía. Cuando la casilla es libre $T[i][j]$ es cierto, y se entra en la sentencia alternativa para añadirle las aristas. La utilidad de este grafo es agilizar la consulta de vecinos.

```
class laberinto {
    bool encontrada;
    int m,n;
    matriz<int> T;
    vector<int> x;
    lista* grafo;
    vector<bool> c;

    bool libre(int i, int j)
        { return 0 <= i && i < m && 0 <= j && j < n && T[i][j]; }

    void crear_grafo() {
        grafo = new lista[m*n];
        for (int i=0; i<m; i++) {
            for (int j=0; j<n; j++) {
                int v = i * n + j;
                grafo[v].crea();
                if (T[i][j]) {
                    if (libre(i,j+1)) grafo[v].pon(v+1);
                    if (libre(i+1,j)) grafo[v].pon(v+n);
                    if (libre(i,j-1)) grafo[v].pon(v-1);
                    if (libre(i-1,j)) grafo[v].pon(v-n);
    }   }   }     }

    void busca(int u) {
        if (u == m*n-1) { encontrada=true; return; }
        c[u] = true;
        para_todo_vecino(l,grafo[u]) {
            if (encontrada) return;
            int v = l→v;
            if (!c[v]) { x[v] = u; busca(v); }
    }   }

public:
    int l;
    laberinto(matriz<int> _T) {
        encontrada = false; T = _T; m = T.m; n = T.n;
        crear_grafo();
        x.crea(m*n,-1);
        c.crea(m*n,false);
        busca(0);
        l = 0;
        if (encontrada) {
            int v = m*n-1;
            while (x[v] != -1) {
                int i = v/n; int j = v%n; T[i][j] = -1;
                v = x[v]; l++;
        }   }
    }
};
```

Algoritmo 7.4 *Clase* laberinto.

Ninguna más. Y por descomptado, no se debe confundir con el grafo implícito de la exploración, que es un árbol dirigido y tiene muchísimos más nodos.

La función miembro *busca()* implementa la búsqueda exhaustiva que nos guiará hasta la solución. Como parámetro de entrada tiene el vértice actual. La salida es el vector de predecesores x, que es variable miembro. El vector de colores del DFS ha sido sustituido por un vector de booleanos c, indicando si el vértice en cuestión ya ha sido considerado, o no. Esta rutina sigue escrupulosamente el Esquema 7.1, excepto alguna modificación debida a que en el problema del laberinto sólo buscamos una solución. En el cuerpo de la función *busca()* hay primero el caso trivial cuando $u = m * n - 1$, como siempre. La parte recursiva es un DFS.

Para calcular la eficiencia de las funciones del Algoritmo 7.4 hay que tener las ideas muy claras. Saber distinguir entre los dos grafos involucrados en la resolución del problema. Sí. Hay dos grafos involucrados. Por un lado un grafo material que representa el laberinto propiamente dicho. Denotando por p al producto $p = m * n$, este grafo tiene p nodos, y más o menos $4 * p$ aristas, ya que cada nodo tiene más o menos cuatro vecinos. Pero por otro lado, tenemos el grafo implícito que representa el árbol de exploración. Este tiene muchísimos más nodos que el anterior. En el árbol de exploración, cada nodo se corresponde con una subsecuencia de decisiones tomadas. Si consideramos que en cada paso tenemos cuatro posibles decisiones a tomar, entonces el árbol de exploración tiene 4^p nodos.

La eficiencia pues, viene dominada por la fuerza bruta. Por la función *busca()*. Siendo un recorrido en profundidad de un grafo, la eficiencia es $\Theta(V + E)$, que como se ha visto es $\Omega(4^p)$, ya que seguro que habrá más aristas que nodos. Por otra parte, el constructor tarda $\Theta(p)$.

7.3 Ramificación y Poda. Optimización

Los algoritmos de backtracking situan los planteamientos de los problemas en una posición diametralmente opuesta a la que se encontraban antes.

Hasta ahora, resolver los problemas consistía en articular mecanismos constructivos. Tanto en los algoritmos voraces, como en la programación dinámica, como en la búsqueda exhaustiva se activan procesos que comienzan sin nada. A partir de cero, se las ingenian para obtener una respuesta a cada instancia de los problemas que solucionan.

Por otra parte, quizás no se ha dicho en palabras textuales, la potencia de cálculo de los esquemas como la programación dinámica o la búsqueda exhaustiva es superior a la de los algoritmos voraces. Con algoritmos voraces, saber que encontrábamos una solución óptima depende del problema. Y para algunos

problemas, la optimalidad depende de los datos de la instancia. Con los de programación dinámica o de búsqueda exhaustiva, siempre se encuentran soluciones óptimas. Una diferencia abismal. La programación dinámica consigue eficiencias polinómicas, pero tiene la desventaja de requerir mucho espacio. La búsqueda exhaustiva obtiene soluciones óptimas con unos requerimientos de espacio razonables. Sin embargo, la búsqueda exhaustiva tiene la desventaja de tardar demasiado. Tenemos todos los problemas de decisión solucionados. Pero tardamos demasiado.

A partir de esta sección trabajaremos a la inversa. Ahora nos dedicaremos a no hacer cosas, más que a hacerlas. Ahora podaremos el árbol de exploración. Por esta razón al principio de esta sección se habla de situar los planteamientos de los problemas en una posición diametralmente opuesta. A partir de ahora, nos dedicaremos a articular mecanismos destructivos. Es lógico, comenzamos calculando diferentes combinaciones de los valores de unas variables de decisión. Entonces alcanzamos un estado en que ya calculamos todas las combinaciones posibles. Pero trabajamos demasiado. Eso provoca que tardemos demasiado tiempo. Llegados a este extremo, hay que dedicarse a ahorrar trabajo para ir más de prisa.

Dejamos el mundo de los problemas decisionales en los que tan solo se busca si existe o no una o varias soluciones, y nos adentramos en el mundo de los problemas de optimización. Ahora las soluciones ya no sólo son simples vectores de decisiones, sinó, además, el valor de la función objetivo para esos vectores. Es notable que con la ramificación y poda, se acostumbra a decir cuánto vale la función objetivo, pero también cuáles son los valores de las variables que producen ese objetivo. Es decir, cuáles son los valores de las decisiones que lo consiguen. En la información que se guarda en los nodos de la exploración hay estos valores de las decisiones, y cada vez que haga falta se calcula el valor de la función objetivo. A los vectores de decisiones los denotaremos por x. A los valores de la función objetivo les llamaremos z. Y a la función que los calcula a partir de un vector de decisiones concreto, $z(x)$.

Si n es pequeño, entonces la técnica del backtracking basta para cualquier problema de optimización... Poco a poco nos vamos introduciendo en una confusión. Tanto tiempo insistiendo en que el backtracking era para los problemas de decisión y la ramificación y poda para los de optimización, y ahora resulta que no. El tamaño de la instancia puede decidir más que el enfoque del problema. Está bien. Confundir el backtracking y la ramificación y poda es legítimo. A veces las palabras son tramposas. La ramificación y poda es una ampliación del backtracking, y al mismo tiempo, un subconjunto. Parece contradictorio, pero es una ampliación porque hace todo lo que hace el backtracking y aún más cosas. Es un subconjunto porque todo algoritmo de ramificación y poda es un algoritmo de backtracking, pero no todo algoritmo de backtracking realiza ramificación y poda.

La característica identitaria de la ramificación y poda es el cálculo de cotas.

El solo hecho de mantener actualizado el mejor valor de la función objetivo obtenido hasta el momento actual ya permite dejar de explorar cuando a media exploración se obtenga un resultado que en ningún caso pueda mejorar el actual. Para saberlo, será preciso calcular cotas. A este criterio de poda, el más primitivo sin duda, se le denomina poda basada en la mejor solución en curso. Es tan famosa, que hay quién lo llama PBMSC. Se denomina *incumbente* a ese valor de la mejor solución en curso.

7.3.1 Cálculo de Cotas

La ramificación y poda se caracteriza por el cálculo de cotas. Ésta es su aportación al esquema de búsqueda exhaustiva. El cálculo de cotas se puede apreciar claramente en el código de los programas. Incluso se puede extraer, convirtiendo fácilmente un algoritmo de ramificación y poda en uno de backtracking.

Una diferencia que salta la vista entre el backtracking y la ramificación y poda es el tiempo de cruzar una arista en el árbol de exploración. En el backtracking es $\Theta(1)$, en la ramificación y poda $\Theta(n)$, ya que es lo que acostumbra a tardar el cálculo de las cotas.

Por otra parte, para el cálculo de cotas se utiliza información relativa a las n decisiones del problema. Y en especial, se utiliza información de las decisiones que quedan por tomar. Eso hace que la parte todavía no asignada de la solución parcial tenga que estar actualizada como tal. En definitiva, se ha acabado guardar las subsoluciones parciales en variables globales. El cálculo de cotas necesita saber qué decisiones todavía no han sido tomadas.

Conviene analizar en rigor esta cuestión. Se ha dicho que entre la información del nodo actual a partir de ahora habrá el vector de subsoluciones. Y pasar un vector por valor no es una cosa que suene demasiado bien. De todas maneras, no hay más remedio, porque el espacio ocupado por las sucesivas copias de estos vectores hacen falta para las cotas. No hay vuelta de hoja.

Todo ello aún va más lejos. La diferencia entre el backtracking y la ramificación y poda, es que los vectores con las subsoluciones se pasen por referencia o por valor. En el backtracking, la parte no resuelta del problema no impacta, y por tanto puede contener cualquier valor. No hay que copiarlo cada vez. En la ramificación y poda la parte no resuelta sirve para el cálculo de cotas, y si retornamos de una subexploración que no ha tenido éxito, hay que restaurar los valores que indican que aquella decisión no ha sido tomada. Al pasar toda la información en la estructura nodo, por valor, la copia de restauración se hace automáticamente gracias al comportamiento recursivo comentado al inicio de este capítulo.

Calcularemos dos tipos de cotas. Por un lado las heurísticas y por otro lado

las relajaciones.

De las heurísticas no hay nada que decir. Simplemente son algoritmos voraces que hay que interpretar como soluciones a ojo. Está claro que cualquier solución óptima, para cualquier problema, será mejor que las soluciones a ojo que podamos calcular. Por tanto, con la heurística ya tenemos una cota. Si tenemos más de una heurística, usaremos la que nos dé la mejor cota, claro.

Y para las cotas del otro lado de la solución óptima, hacen falta las relajaciones.

Relajaciones

Relajar un problema significa ignorar algunos requerimientos. Jugar a ser dios. Si una solución debe cumplir las condiciones α y β, entonces resolviendo óptimamente el problema para aquellas soluciones que safisfagan α, encontraremos una solución que será mejor o igual a la de nuestro problema. La acotará.

Reflexionemos. Tenemos un tipo de problemas cuya solución es una combinación de los posibles valores de los elementos de entrada que maximice una cierta función. La solución puede ser expresada en un solo número, el valor máximo de la función dentro de las limitaciones que imponen las condiciones. Tenemos algoritmos enumerativos capaces de calcular todas las posibles respuestas del problema en función de todas las posibles combinaciones de los valores de los elementos de entrada. Pero tardan demasiado. Idea brillante, si para resolver estos problemas podemos encontrar cotas inferiores y superiores en tiempos razonables, vamos por el buen camino. Para el caso de un problema de maximización, un algoritmo voraz nos dará una cota inferior de la solución óptima. O sea, si resolvemos el problema a ojo, entonces la solución óptima será como mínimo tan buena como la nuestra. Por otro lado, si relajamos el problema, obtendremos una cota superior, ya que si pasamos olímpicamente de respetar algunas condiciones, entonces seguro que el valor de la mejor solución que obtengamos será mejor o igual a la del problema inicial con todas sus restricciones. En definitiva, si tenemos algoritmos polinómicos par acotar inferior y superiormente el valor de una solución óptima, entonces tenemos el problema de optimización combinatoria resuelto polinómicamente en el momento en que seamos capaces de hacerlas coincidir. Y si no somos capaces de eso, entonces, al menos, las dos cotas ahorrarán gran parte de la exploración.

7.3.2 Esquema Algorítmico de Ramificación y Poda

Sin duda, el que más relevancia de los esquemas algorítmicos iluminados en este discurso ha tenido en la sociedad, es el que se presenta a continuación.

El Esquema 7.2 es para un problema de minimización.

```
algoritmo branch&cut(p)
{
    si (p.i = n) {
        si (p.z < z) {
            x ← p.x
            z ← p.z
        }
    }
    sino {
        para v = cada valor posible de x_{i+1} {
            x_{i+1} ← v
            nodo q ←arista(p, x_{i+1})
            q.z = z(q.x)
            q.z_ = calcula_cota(q)
            si (factible(q.x) i q.z + q.z_ < z) {
                q.i ← p.i + 1
                branch&cut(q)
            }
        }
    }
}
```

Esquema 7.2 *Ramificación y poda para un problema de minimización.*

El parámetro que se recibe por valor es una estructura que llamamos *nodo*. Esta estructura contiene el vector completo con la subsolución actual $p.x$, el valor real $p.z$ asociado a este vector, la cota inferior (problema de minimización) de la función objetivo, $p.z\_$, y el índice de la decisión actual $p.i$, que como antes, es el paso de recursividad y la dimensión de la solución parcial $p.x$, válida.

Tanto la incumbente z con el valor óptimo de la función objetivo, como el vector de variables enteras x, con la combinación de decisiones donde se produce, se guardan en variables de visibilidad global, o variables miembro de la clase que se implementa.

En el Esquema 7.2, la poda es la expresión $q.z + q.z\_ < z$. Eso significa que si lo que tenemos hasta ahora más lo mejor que podamos conseguir es peor que la mejor solución que hayamos encontrado, entonces dejamos de explorar por esta rama.

La ramificación y poda es amiga de los algoritmos voraces. La programación dinámica es un mundo aparte. Antes de llamar al Esquema 7.2 se utiliza un algoritmo voraz que se puede decir que encuentra una solución inicial a ojo. Eso nos permite inicializar la incumbente, o sea la solución actual.

Con todo, *calcular_ cota()* concentra nuestro punto de entrada. Dedicaremos todos nuestros esfuerzos en refinar la calidad de las cotas, ya que cuanto más ajustadas sean, menos tiempo se tardará en explorar el subárbol resultante.

El cálculo de cotas, desde un punto de vista filosófico, es mucho más que una simple rutina. Es un motor de conocimiento. No sólo puede incluir otras ideas voraces, sinó que permite, a medida que pase el tiempo y nos hagamos más sabios, ir recortando el tiempo de solución. Volviendo al juego del ajedrez, el cálculo de cotas representa una vía para introducir conocimiento adicional al programa. Por ejemplo, supongamos que alguien nos dice "En el juego de ajedrez, no conviene que te maten la reina". Entonces el algoritmo puede tomar nota. Todos aquellos vértices del árbol de exploración que representen posiciones en las que el adversario amenaza nuestra reina pueden ser evitados. Si suponemos que hacemos un movimiento y la respuesta que obtenemos nos mata la reina, entonces descartamos el movimiento y no seguimos explorando el subárbol donde nos conduzca esa decisión.

7.3.3 El Problema de la Asignación

El problema de la asignación es un problema bastante abstracto que se ajusta a multitud de situaciones de la vida cotidiana. Se trata de seleccionar parejas de elementos de un universo formado por dos conjuntos. Las parejas están compuestas por un elemento de cada conjunto. Para asignar cada pareja posible tenemos un coste. El problema consiste en asignar las n parejas con el mínimo coste. Aquí, se formaliza el problema utilizando los conceptos de proyecto y empresa.

Definición 7.4 Problema de la Asignación. *Dados n proyectos, n empresas, y la matriz de costes $n \times n$ con lo que cobra cada empresa por realizar cada proyecto, asignar un proyecto a cada empresa minimizando el coste total.*

Así pues, el problema de la asignación es un problema de minimización. En el análisis siguiente se tiene en cuenta en todo momento.

La forma de trabajar del algoritmo de ramificación y poda con el problema de la asignación se ilustra en el ejemplo siguiente. Se trata de una instancia con $n = 4$. Las empresas se llaman *a, b, c,* y *d.* Los proyectos, *A, B, C,* y *D.*

La matriz de lo que cobra cada empresa por realizar cada proyecto se muestra en la Tabla 7.1.

El primer paso de los algoritmos de ramificación y poda es calcular una solución heurística con un algoritmo voraz. Para los problemas de minimización, esto dará una cota superior de la incumbente. Así, de entrada, todas las soluciones posibles que antes de ser calculadas completamente ya sean más caras

	A	B	C	D
a	11	12	18	40
b	14	15	13	22
c	11	17	19	23
d	17	14	20	28

Tabla 7.1: *Coste de las empresas a,b,c y d, por los proyectos A,B,C y D.*

que la voraz podrán ser descartadas. Este cálculo inicial de la incumbente sólo es útil mientras la exploración no encuentre ninguna solución completa mejor. Cada vez que se mejore el valor de la solución encontrada, la incumbente, se actualiza también la cota superior, siendo un problema de minimización.

En el ejemplo, la asignación voraz podría ser $(a \to A, b \to B, c \to C, d \to D)$. Si llamamos z^* al valor de la incumbente, o mejor solución encontrada, tenemos $z^* \leq 11+15+19+28 = 73$. Eso representa una cota superior. La peor solución que el algoritmo puede retornar, de momento.

Por otro lado, hay que establecer la forma de calcular las cotas inferiores en cada nodo. Cotas inferiores para un problema de minimización sólo pueden encontrarse relajando restricciones. Para el caso que nos ocupa tenemos dos posibilidades, ya que tenemos dos tipos de restricciones.

- Cálculo de la cota inferior ignorando empresas. Si pudiéramos no asignar ningún proyecto a las empresas más caras (o sea, no contratarlas), y asignar dos proyectos a las que lo hacen mejor de precio (contratándolas dos veces para poder asignar todos los proyectos), entonces obtendríamos una solución probablemente no factible, pero de valor mejor o igual que la del problema que efectivamente se plantea. El cálculo de la cota inferior de z^* es el mínimo coste de cada proyecto, o sea la suma de los mínimos de cada columna de la Tabla 7.1, $z^* \geq 11 + 12 + 13 + 22 = 58$. Notad que respetamos, eso sí, la condición de tener que realizar todos los proyectos.
- Cálculo de la cota inferior ignorando proyectos. Si pudiéramos no asignar ninguna empresa a los proyectos más caros (o sea, no hacerlos), y asignar un mismo proyecto a dos empresas diferentes cuando cuesten poco (haciéndolos dos veces para contratar todas las empresas), entonces, como antes, obtendríamos una solución probablemente no factible, pero de valor mejor o igual que la de nuestro problema. El cálculo de la cota inferior de z^* es el mínimo coste de cada empresa, que significa la suma de los mínimos de cada fila en la Tabla 7.1. Por tanto, $z^* \geq 11 + 13 + 11 + 14 = 49$. En este caso cumplimos la condición de asignar algún proyecto a cada empresa.

De las dos relajaciones se cogerá la que más convenga. Se están buscando cotas inferiores. Cuanto mayores sean, mejor. O sea, cogemos la primera. Por tanto, en este momento ya se puede asegurar que $58 \leq z^* \leq 73$. Con estas dos cotas, se inicia el algoritmo enumerativo.

En la Figura 7.6 se puede ver el estado de la exploración habiendo hecho el primer anidamiento. Los recuadros en tono claro indican valores de cotas inferiores calculadas en cada nodo. Para el caso primero se descompone la suma. Como ya sabemos que $z^* \leq 73$, se puede podar la rama de la derecha $(a \rightarrow D)$, ya que tiene una cota inferior, 78, peor que la solución actual, 73. O sea, superior. No es de estrañar. La empresa a cobra una barbaridad para realizar el proyecto D.

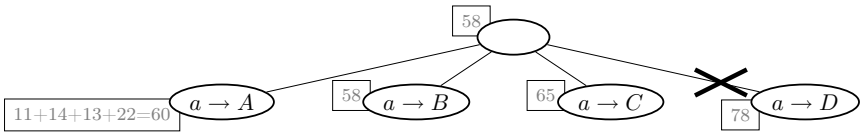


Figura 7.6: *Estado de la exploración después del primer nivel de anidamiento.*

Se denomina criterio de *branching* al que se utiliza para la selección de qué rama de las abiertas merece la pena explorar. El branching del código que acompaña el libro simplemente coge la siguiente rama abierta, en todo momento. En este ejemplo sin embargo, se explora siempre la rama más prometedora. Ahora pues, se procede explorando la rama $(a \rightarrow B)$, con cota 58. El nuevo estado se muestra en la Figura 7.7.

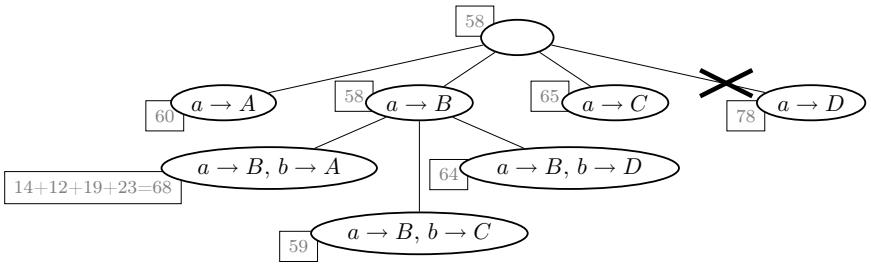


Figura 7.7: *Estado de la exploración después del segundo nivel de anidamiento.*

Atención al tema de las cotas. Cada vez que se testea una nueva decisión, todos los nodos descendientes de ésta han de calcular la cota inferior habiendo fijado los valores correspondientes a las decisiones ya tomadas. Es decir, las dos relajaciones que se han visto en la página anterior se correspondían sólo con el instante inicial, previo a la exploración. Allí se decía el mínimo de todas las columnas. Aquí, la cota inferior es la suma de los costes de los proyectos asignados más la suma de los mínimos costes de los proyectos no asignados. Y para la segunda relajación igual, pero con las filas o empresas.

Después de la primera decisión $(a \rightarrow B)$, niguna rama tiene cota inferior peor que la incumbente $z^* = 73$. No se puede descartar ninguna. Seguimos pues por la rama más prometedora, la $(a \rightarrow B, b \rightarrow C)$ con cota inferior 59. Se

alcanza el estado que se muestra en la Figura 7.8.

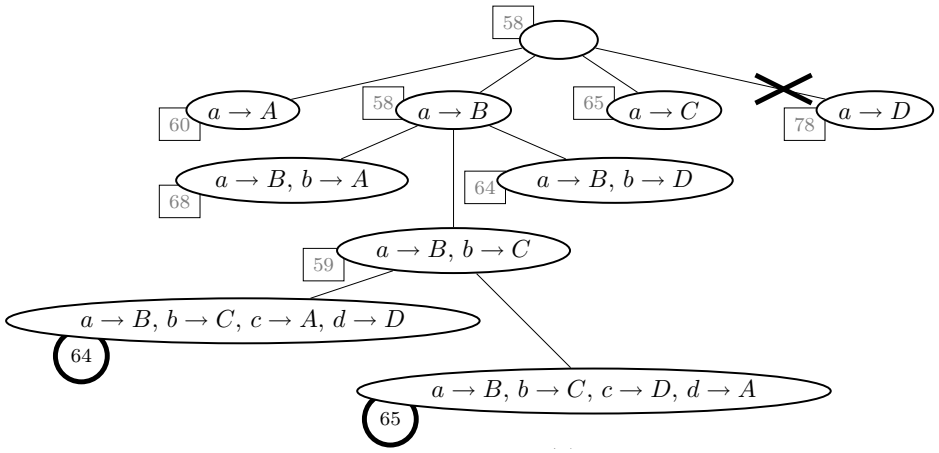


Figura 7.8: *Estado de la exploración después del tercer nivel de anidamiento.*

Llegados a este punto, se han conseguido dos soluciones completas, y por tanto factibles. De las dos, la mejor es la $(a \to B, b \to C, c \to A, d \to D)$ que da un valor de la función objetivo igual a 64. En la Figura 7.8 aparece en un círculo para diferenciarse de las cotas. Ahora ya no se trata de una cota sinó de un valor posible de la función objetivo. Por tanto, sustituye el valor de la incumbente que valía 73 y partir de este momento vale $z^* = 64$.

Ya se puede afirmar que el valor óptimo de la solución quedará entre $58 \leq z^* \leq 64$.

Además, eso también permite podar algunas de las ramas que en este momento hay abiertas. Las aspas que aparecen cortando las ramas del árbol en la Figura 7.9 significan podas, y por tanto no serán exploradas porque como mucho se obtendría una solución igual de buena que la que ya se tiene. Sin embargo, es de sentido común que si se trata de encontrar todas las soluciones óptimas posibles, entonces no podaríamos cuando el valor de la cota inferior fuese igual a la incumbente, sólo cuando sea estrictamente superior.

En la Figura 7.9 ya tan solo queda una rama por explorar. La rama de la izquierda, $(a \to A)$, todavía puede dar alguna solución mejor que la obtenida hasta ahora. La exploración continua por esa rama. Después de ejecutarse tres veces seguidas el caso trivial, o sea, retrocediendo en el árbol de exploración primero, y de dos niveles de anidamiento, o sea profundizando después, en la rama $(a \to A)$ se llega al estado que se muestra en la Figura 7.10.

Se puede ver que una vez se ha profundizado por la decisión $(a \to A)$, en un primer nivel se han podido podar dos de sus tres sucesores por tener cotas inferiores mayores a la incumbente $z^* = 64$. En cambio, al explorar la única

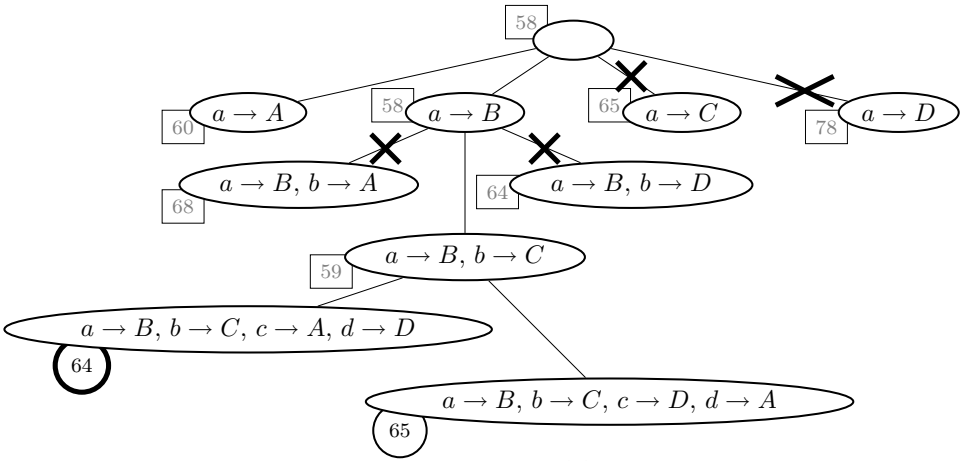


Figura 7.9: *Podas resultantes de la nueva solución* $z^* = 64$.

rama candidata que quedaba, se encuentra la solución óptima $z^* = 61$, con la asignación $(a \to A, b \to C, c \to D, d \to B)$. Se puede garantizar que es la asignación óptima porque no quedan ramas por explorar, y por tanto, se han comprobado todas las demás.

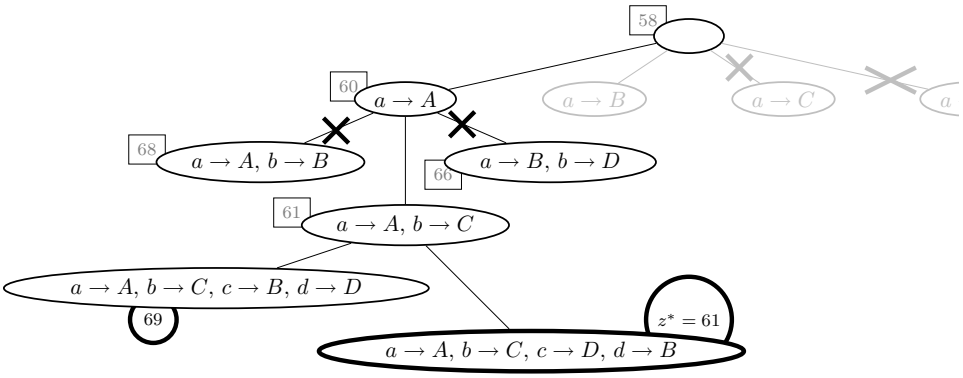


Figura 7.10: *Solución óptima* $z^* = 61$.

Es interesante observar una cierta falta de simetría. En los problemas de minimización, por un lado, las cotas superiores se obtienen a partir de soluciones factibles, y no dependen del nodo de exploración donde se esté. Por el otro, las cotas inferiores se obtienen a partir de soluciones no factibles, y se calculan en cada nodo de la exploración.

Bien, vamos a por la implementación. Desafortunadamente, como los problemas que se resuelven son cada vez más complejos, se ha llegado a un punto en el cuál una clase completa para resolverlos no cabe en una sola página.

En el Algoritmo 7.5 se muestra la declaración de la clase, el constructor y un par de funciones booleanas muy simples, útiles para el cálculo de cotas.

Las variables miembro de la clase *asignacion* implementada aquí son las preceptivas de los problemas de ramificación y poda. Como privadas, aquellas que sirven para guardar los datos de la instancia que se resuelve. El número de empresas o proyectos, n, y la matriz de costes empresa proyecto, C.

También se declara una estructura privada que contiene la información asociada a un nodo en el árbol de exploración. La estructura *nodo* contiene estos datos:

- Nivel de decisión en el que nos encontramos i, como en el backtracking.
- Valor de la función objetivo z para la subsolución actual.
- Valor de la cota inferior $z\_$ que nos indica cuál es el mejor valor que podríamos obtener incluyendo las decisiones no tomadas aún, la parte que todavía no se ha explorado.
- Vector con la asignación parcial a, que hace el papel del campo x de la estructura nodo del Esquema 7.2. En esta asignación, para cada empresa e guarda el proyecto que tiene asignado $a[e]$.

Como variables miembro públicas se dejan tan solo las que el programa cliente puede requerir. Eso es, el valor de la solución óptima z, que también utilizaremos de incumbente, y el argumento donde se ha producido, x.

Como funciones miembro, todas ellas privadas, hay la heurística *voraz()* para la cota superior, que no retorna nada porque actualiza directamente la incumbente.

A la rutina *cota(a)* se le pasa una asignación que no modificará. Esta asignación se pasa por referencia porque es un vector, pero funcionalmente es un paso de parámetro por valor. Calculará la cota inferior, lo mejor conseguible, respetando las decisiones ya tomadas en la asignación que se le pasa.

El procedimiento *arista()* sirve para transitar de un nodo de la exploración a otro. Eso significa calcular cada uno de los campos de la nueva estructura *nodo*. Para eso, se le pasa por valor el nodo actual, el indicador de la arista que hay que seguir, y por referencia, para que lo rellene, el nodo siguiente.

La última función privada es el branch & cut, que se le llama *busca()*.

```
class assignacion {
    int n;
    matriz<double> C;

    struct nodo {
        int i;
        double z;
        double z_;
        vector<int> a;
        nodo(int n = 0) {
            i = 0;
            z = 0.0;
            z_ = 0.0;
            a.crea(n,-1);
        }
        simnodo() { a.destruye(); }
    };

    void voraz();
    double cota(vector<int> a);
    void arista(nodo s, int i, nodo& t);
    void busca(nodo s);

    bool empresa_libre(int e, vector<int> a) { return a[e] == -1; }

    bool proyecto_libre(int p, vector<int> a) {
        for (int e=0; e<a.n; e++) if (a[e] == p) return false;
        return true;
    }

public:
    double z;
    vector<int> x;
    asignacion(matriz<double> _C) {
        C = _C;
        n = C.n;
        x.crea(n,-1);
        nodo s(n);
        s.z_ = cota(x);
        vorac();
        busca(s);
    }
    ~asignacion() { x.destruye(); }
};
```

Algoritmo 7.5 *Declaración de la clase* asignacion.

Dentro de la misma declaración de la clase *asignacion* se añaden un par de funciones booleanas a las que se les pasa una asignación parcial. En función de esta asignación, la rutina *empresa_libre()* retorna si una determinada empresa ya tiene algún proyecto asignado, o no, en $\Theta(1)$. Y la otra, la función *proyecto_libre*, comprueba lo mismo para los proyectos en $\Theta(n)$. Este par de procedimientos se utilizan en el cálculo de cotas, como se ve a continuación.

El constructor toma nota de los parámetros, inicializa el vector solución, crea el nodo inicial y le da una cota inferior. Después calcula la solución heurística para actualizar la incumbente, y arranca la ramificación y poda. La eficiencia de todo ello vendrá dominada por el branch & cut.

La primera de las cuatro funciones que observamos es el algoritmo voraz por ser el procedimiento más independiente desde un punto de vista funcional. En el Algoritmo 7.6 hay la implementación. La idea astuta que implementa es asignar cada empresa a un proyecto de la manera más fácil posible respetando la factibilidad. Así obtenemos una cota superior inicial al problema con la suma de los elementos de la diagonal de la matriz de costes. Si este algoritmo fuese más sabio, entonces la exploración tardaría menos.

```
void voraz() {
    z = 0.0;
    for (int u=0; u<n; u++) {
        z = z + C[u][u];
        x[u] = u;
    }
}
```

Algoritmo 7.6 *Algoritmo voraz para la cota superior inicial del problema de la asignación (minimización).*

Esta aproximación greedy requiere $\Theta(n)$.

En el Algoritmo 7.7 se muestra la función *cota()* para el cálculo de cotas en cada nodo. Como se ha dicho más arriba, se calculan dos cotas inferiores según dos relajaciones del problema. Con una sola ya podría funcionar todo ello, pero siempre es mejor afinar las cotas tanto como sea posible. Así pues, las dos partes del Algoritmo 7.7 son completamente independientes. Al final, se retorna el mejor de los dos valores obtenidos. La función recibe una asignación como parámetro de entrada que representa una solución parcial al problema, y por tanto en ella hay todavía empresas y proyectos libres.

El procedimiento comienza ignorando que un proyecto sólo pueda ser asignado a una empresa. Que el mismo proyecto pueda ser asignado a diferentes empresas, permite asignar a cada empresa libre su proyecto más barato, de los

proyectos que aún estén libres. Así, se ignora que el mismo proyecto ya se le haya atribuido a alguna otra empresa. Estamos calculando la cota de tal manera que quizás estemos suponiendo que todas las empresas haran el mismo proyecto, que, por casualidad, fuera un proyecto muy barato independientemente de quien lo realice. Entonces la cota inferior que obtendremos, *ze* en el algoritmo, será la suma de los costes de los proyectos libres más baratos que puedan ser asignados a cada empresa libre.

```
double cota(vector<ynt> a) {
    double ze = 0;
    for (int e=0; e<n; e++) {
        if (empresa_libre(e,a)) {
            double me = oo;
            for (int p=0; p<n; p++) {
                if (proyecto_libre(p,a)) {
                    if (me > C[e][p]) me = C[e][p];
                }
            }
            ze = ze + me;
        }
    }

    double zp = 0;
    for (int p=0; p<n; p++) {
        if (proyecto_libre(p,a)) {
            double mp = oo;
            for (int e=0; e<n; e++) {
                if (empresa_libre(e,a)) {
                    if (mp > C[e][p]) mp = C[e][p];
                }
            }
            zp = zp + mp;
        }
    }
    return max(ze,zp);
}
```

Algoritmo 7.7 *Relajación para las cotas inferiores del problema de la asignación.*

Por otro lado, en la segunda parte, se utiliza la otra relajación. Esta vez se trata de suponer que cada proyecto será hecho por la empresa que cobre menos por hacerlo. Entonces la cota inferior que obtendremos, *zp* en el algoritmo, será la suma de los costes más baratos de cada empresa libre por hacer los proyectos que queden libres.

La función *cota()* del Algoritmo 7.7 tiene una eficiencia peor que las anteriores. Es $\Theta(n^3)$, debido a la primera parte del procedimiento, ya que tenemos un tratamiento $\Theta(n)$ dentro de un par de bucles anidados que la enmarcan en $\Theta(n^2)$. La segunda parte es más rápida, ya que la función *empresa_ libre()* es $\Theta(1)$ y no $\Theta(n)$ como la *proyecto_ libre()* de la primera parte. Para agilizar la eficiencia global del algoritmo se podría mantener un vector de proyectos libres en cada nodo.

Provistos de la cota superior y del cálculo de las cotas inferiores, ya tan solo queda implementar la exploración. En la solución que aquí se presenta, se ha asociado cada decisión de la ramificación y poda con la asignación de un nuevo proyecto a alguna empresa disponible. Es decir, cada decisión se corresponde a un proyecto, y cada valor que puede tomar la decisión, a una empresa.

```
void arista(nodo s, int e, nodo& t) {
    t.i = s.i + 1;
    t.z = s.z + C[e][s.i];
    t.a = s.a;
    t.a[e] = s.i;
    t.z_ = cota(t.a);
}
```

Algoritmo 7.8 *Cambio de nodo en el grafo de exploración para el problema de la asignación.*

En el Algoritmo 7.8 se hacen las asignaciones necesarias para cambiar de nodo. A partir de los parámetros de la cabecera de la función *arista()* ya se puede intuir que estamos en el nodo s, y asignándole el proyecto $s.i$ a la empresa e pasaremos a un nuevo nodo t en el árbol de exploración.

Este procedimiento comienza incrementando el paso de exploración, i, en la copia al nuevo nodo. Este índice también se corresponde con el número de proyectos asignados en la subsolución, o sea, en el nodo actual de exploración. Entonces se calcula el valor de la función objetivo obtenida en este nuevo nodo t de manera incremental, utilizando el parámetro e que representa la empresa a la que se le asigna el nuevo proyecto.

En la implementación de la estructura *vector* del Algoritmo 7.1, ya se decía que el operador de asignación hace la copia completa del contenido. Eso es importante en este punto. Al ser vectores, como parámetros se pasan por referencia. No obstante, la copia $t.a = s.a$ debe ser de todos los valores y no de la dirección. Si la copia fuese por referencia, o sea, si tan solo se copiara la dirección de los vectores y todos los nodos operasen sobre un mismo espacio físico, entonces las tentativas fallidas de la exploración interferirían en el cálculo de las cotas de los nodos posteriormente explorados. Por otro lado, este mismo

hecho provoca que la eficiencia de la función *arista()* sea $\Theta(n)$, aunque no lo parezca.

La implementación de la clase *asignacion* culmina con el branch & cut. En el Algoritmo 7.9 se muestra el procedimiento que realiza la exploración. Tanto en el caso trivial como en los casos recursivos, el procedimiento se ciñe al Esquema 7.2 hasta el último detalle.

```
void busca(nodo s) {
    if (s.i == n) {
        if (s.i < z) {
            z = s.z;
            x = s.a;
        }
        return;
    }
    for (int p = 0; p < n; p++) {
        if (empresa_libre(p,s.a) && s.z + s.z_ < z) {
            nodo t;
            arista(s,p,t);
            busca(t);
        }
    }
}
```

Algoritmo 7.9 *Ramificación y poda para el problema de la asignación.*

No hablemos de su eficiencia. Dejémoslo correr. Conviene fijarse en que en todos los casos el árbol de exploración tiene profundidad n, y en cada nivel de recursividad tenemos n posibles nodos sucesores. Eso significa que el árbol tiene n^n hojas. En alguna o algunas de ellas hay la solución óptima del problema. Si no hay demasiada suerte y la poda no resulta lo bastante efectiva, para problemas medianamente grandes, estas empresas se morirían de hambre.

7.3.4 El Problema del Viajante

En inglés, *The Traveling Salesman Problem*, o directamente TSP, es una referencia como punto fronterizo del conocimiento humano. Normalmente se plantea como un viajante de negocios que debe pasar por n ciudades. Entonces los datos del problema consisten en la matriz de distancias entre esas ciudades. Aquí lo definiremos de una manera más formal.

Definición 7.5 El Problema del Viajante. *A partir de una función real no negativa de costes definida sobre las aristas del grafo no dirigido completo de n*

vértices, $c: E(\mathcal{K}_n) \to \mathbb{R}^+ \cup \{0\}$*, encontrar un ciclo simple de mínimo coste que contenga todos los nodos.*

Las soluciones del TSP son selecciones de n aristas del grafo completo de n vértices, $\mathcal{K}_n$.

El TSP es uno de los problemas abiertos más importante desde hace casi cinco décadas. La mejor solución polinómica fue propuesta el 1976 por Christofides [7], consiguiendo una aproximaxión al óptimo de $3/2$, que significa dar una solución un 50% más cara que la óptima. O sea que nos queda mucho camino por recórrer. Quizás os preguntaréis cómo puede saberse que la solución dada es a $3/2$, como mucho, de la óptima. No es difícil de entender. Hay cotas. O sea, si sabemos que como mucho el óptimo puede ser diez y obtenemos quince, entonces podemos asegurar que estamos a menos de $3/2$ del óptimo. Profusa literatura relativa al TSP ha sido publicada. Obras monográficas muy extensas son [2, 7, 15, 18].

Hoy día hay mucho movimiento respeto al tema del TSP. En la Figura 7.11 se muestra una solución bastante espectacular, que todavía no es óptima, para una instancia planteada con $n = 100000$ ciudades, distribuidas dibujando la Mona Lisa, [5]. En la página web correspondiente se pueden consultar concursos y otras historias relativas al problema.

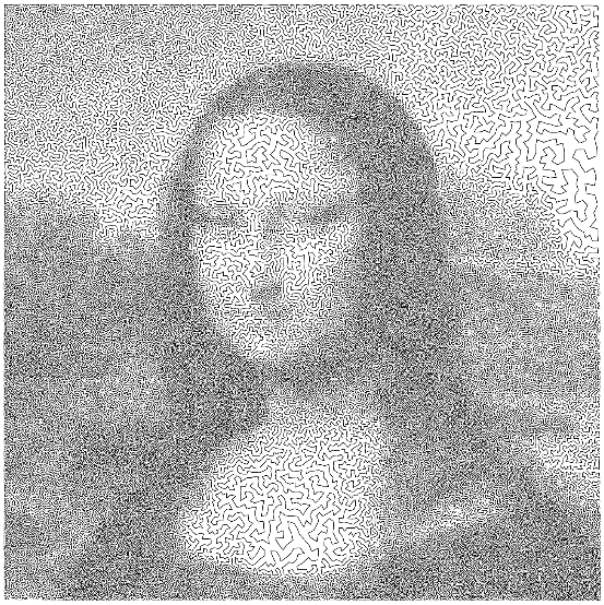


Figura 7.11: *Solución a una instancia del TSP.*

Como efectivamente se puede comprobar en la imagen, las soluciones a instancias reales del TSP no acostumbran a tener ciclos. Son líneas que se enroscan y zigzaguean rellenando la superficie donde se dibujan, pero sin cruzarse.

En esta sección se presenta en primer lugar una implementación de la clase *viajante* que utiliza un algoritmo enumerativo. Se sigue con una introducción a los modelos poliédricos en un enfoque meramente ilustrativo. Y cierra el capítulo un análisis poliédrico del problema del viajante.

La clase *viajante*

En el Algoritmo 7.10 hay la declaración de una clase para resolver el problema del viajante. Es similar a la implementada para el problema de la asignación en la Sección 7.3.3.

De variables miembro privadas hay las del esquema. O sea, el tamaño de la instancia, n, y la matriz de distancias D que viene dada por la instancia a resolver, de $n \times n$. Además, también se utilizará un vector de cotas inferiores *ci* para calcular la cota inferior en cada nodo de la exploración.

Seguidamente se define la estructura *nodo* que representa el nodo actual del árbol de exploración. Consta de los campos estos:

- Nivel de decisión en el que nos encontramos i que coincide con la longitud de la ruta seleccionada hasta ahora.
- Identificador v del último vértice del grafo incluido en la subsolución.
- Valor del coste de la ruta actual z.
- Valor de la cota inferior $z\_$ que nos indica cuál es el mínimo coste previsible de la parte restante.
- Vector con la ruta parcial actual p, que hace el rol del campo x de la estructura nodo del Esquema 7.2. Este vector contiene el predecesor de cada nodo presente en la subsolución actual, y menos unos para los que todavía no se han visitado.

Después se declaran las funciones miembro privadas, que son exactamente las que corresponden a una clase para implementar un algoritmo de ramificación y poda. Es decir, éstas:

- Heurística para la cota superior inicial. Superior porque es un problema de minimización.
- Función de cotas que se llamará desde cada nodo de la exploración.
- Función que nos materializa la transición entre nodos.
- Función que en definitiva hace la exploración.

```
class viajante {
      int n;
      matriz<double> D;
      vector<double> ci;

      struct nodo {
            int i;
            int v;
            double z;
            double z_;
            vector<int> p;
            nodo(int n) {
                  i = 0; v = 0; z = 0.0; z_ = 0.0;
                  p.crea(n,-1);
            }
      };

      void voraz();
      double cotas();
      void arista(nodo s, int v, nodo& t);
      void busca(nodo s);

public:
      double z;
      vector<int> x;

      viajante(matriz<double> _D) {
            D = _D;
            n = D.n;
            ci.crea(n,oo);
            x.crea(n,-1);
            voraz();
            nodo s(n);
            s.z_ = cotas();
            busca(s);
      }
};
```

Algoritmo 7.10 *Declaración de la clase* viajante.

Como variables públicas se declaran las que guardan la solución del problema, z para el valor óptimo de la función objetivo, y x para almacenar el vector de predecesores donde se produce el óptimo z.

El constructor del Algoritmo 7.10 recibe como parámetro de entrada la matriz de distancias, que ha de ser simétrica aunque no se comprueba. Toma nota de esos parámetros e inicializa el vector de cotas inferiores a infinito, ya que se

calcularán como mínimos. Después inicializa la solución estableciendo todos los vértices como no visitados, y calcula la cota superior inicial con el algoritmo greedy. Finalmente crea el nodo inicial de la exploración, le actualiza la cota inferior global, y arranca el branch & cut.

Pasando rápidamente la vista sobre el Algoritmo 7.11, se ve que la idea heurística brillante implementada, que teóricamente debería servir para podar cuanto más mejor, no hace más que encadenar los vértices uno tras otro. No se luce demasiado. De hecho, es una porquería de solución voraz, igual que la del problema de la asignación. La única ventaja que tienen estas dos rutinas es la legibilidad. Únicamente están aquí para ilustrar la técnica, ya que funcionalmente son totalmente inútiles. A la hora de la verdad no ayudan nada de nada, ya que no hacen más que la primera rama explorada en el branch & cut. Pero bien, el propósito del texto es didáctico, y todo ello es correcto para describir la metodología.

```
void voraz()
{
    z = 0.0;
    for (int u=1; u<n; u++) {
        z = z + D[u-1][u];
        x[u] = u-1;
    }
    z = z + D[n-1][0];
    x[0] = n-1;
}
```

Algoritmo 7.11 *Algoritmo voraz para la cota superior del problema del viajante.*

En el Algoritmo 7.11 es interesante observar que después de secuenciar los $n-1$ primeros vértices se añade la última arista ya que la solución ha de ser un ciclo. Esto hará que la exploración acabe en el nivel $n-1$. La eficiencia de la función *voraz()* pertenece a $\Theta(n)$.

Para calcular las cotas del trozo de ruta restante en cada nodo de la exploración se precisa alguna relajación del problema que nos proporcione cotas inferiores al valor de la solución. Algún número que mejor que eso sea imposible. No es demasiado difícil imaginar que como mínimo, la ruta solución valdrá la suma de los costes de las aristas más baratas incidentes en cada vértice. Es decir, como seguro que hay que pasar por todos los vértices, como mínimo se llegará a ellos por la arista más barata. Calculemos pues la suma de las distancias entre cada vértice del grafo y su vecino más cercano. La relajación ha consistido en no respetar las restricciones de conectividad de la ruta resultante.

```
double cotas()
{
      double z = 0;
      for (int u=0; u<n; u++) {
          for (int v=0; v<n; v++) if (v!=u) {
              if (ci[u] > D[u][v]) ci[u] = D[u][v];
          }
          z = z + ci[u];
      }
      return z;
}
```

Algoritmo 7.12 *Relajación para las cotas inferiores del problema del viajante.*

Es importante comprender la gestión de las cotas inferiores. Para cada nodo nos guardamos la distancia a su vecino más cercano. Eso significa que en el caso ideal, ésta sería la arista que pertenecería a la solución. Después, en la exploración, no habrá más remedio que tocar de pies al suelo, y si para un nodo esperábamos que se pudiese is a su vecino más cercano y no ha sido así, entonces deberemos actualizar la cota total, quitándole lo que realmente se ha gastado para visitar este nodo, y añadiéndole su cota inferior. Por ejemplo, pensábamos que el viaje total costaría 10, y que para pasar por este nodo gastaríamos 1 de esas 10. Pero hemos gastado 2. O sea, que a la cantidad total que pensábamos, 10, le añadimos 2 y le quitamos el 1 que era nuestra cota inferior. De manera que el sesgo entre las espectativas y la realidad irá augmentando a medida que se profundice en la exploración. Bien, de momento, en el Algoritmo 7.12 nos guardamos el vector con las distancias a los vecinos más cercanos de cada nodo. Y además, retornamos la suma de todos ellos. Todo ello representa $\Theta(n^2)$.

El procedimiento que establecerá los nuevos valores después de una transición entre nodos del árbol se muestra en el Algoritmo 7.13.

```
void arista(nodo s, int v, nodo& t)
{
     t.i = s.i + 1;
     t.v = v;
     t.z = s.z + D[s.v][t.v];
     t.z_ = s.z_ - ci[t.v];
     t.p = s.p;
     t.p[t.v] = s.v;
}
```

Algoritmo 7.13 *Cambio de nodo en el grafo de exploración para el problema del viajante.*

Los parámetros son el nodo actual de la exploración s, el vértice que añadimos a la solución parcial, v, y el nodo resultante en el árbol de exploración t. Se comienza incrementando el nivel del nodo en el que nos encontramos del árbol de exploración cuando se hace la copia del campo i, de s a t. Este entero también significa la longitud de la subsolución actual que se guarda en el vector de predecesores p. Se establece v como último vértice de la ruta en el nodo t. Se actualiza el valor de la función objetivo incrementalmente, vale lo que valía en el nodo s, más el coste de la arista que se está añadiendo a la solución, $D[s.v][t.v]$. Entonces hay la actualización de la cota. Eso es lo que se ha mencionado más arriba referente a tocar de pies al suelo. La cota, optimista, esperaba que para salir del nodo v lo hiciéramos por la arista más barata, $ci[t.v]$, y en cambio, lo hemos hecho por una que cuesta $D[s.v][t.v]$. Bien, puede ser que sea la misma. En ese caso, los valores de la función objetivo y de la cota se mantendrían invariables. Eso equivaldría a decir que en la siguiente iteración, lo más probable es que aún merezca la pena seguir este camino de exploración. Finalmente, hacemos la copia de los valores de la solución parcial que teníamos hasta el momento s, y le añadimos la última arista dejando constancia de que el predecesor del nodo recién introducido en la solución, $t.v$, es el último de los que había hasta ahora, $s.v$.

La eficiencia de la función de l Algoritmo 7.13 viene dominada por la copia del vector de predecesores, $\Theta(n)$.

Y ahora que ya le tenemos el pie en el cuello, rematamos el problema con la función que hará la exploración.

```
void busca(nodo s) {
    int u = s.v;
    if (s.i == n-1) {
        s.p[0] = u;
        s.z = s.z + D[0][u];
        if (s.z < z) {
            x = s.p;
            z = s.z;
        }
        return;
    }
    for (int v=1; v<n; v++) {
        if (s.p[v] == -1 && s.z + s.z_ < z)) {
            nodo t;
            arista(s,v,t);
            busca(t);
        }
    }
}
```

Algoritmo 7.14 *Ramificación y poda para el problema del viajante.*

En el Algoritmo 7.14 se muestra una implementación. Otra vez se sigue meticulosamente el Esquema 7.2, con alguna ligera modificación.

Por un lado, en el caso trivial, acabamos cuando la profundidad del árbol es $n-1$, ya que la solución ha de ser un ciclo, y llegados a este punto no nos queda ningún margen de liberdad. Por ese motivo, añadimos la arista hasta el primer vértice que cierra el ciclo antes de considerar el valor de la función objetivo final. Y entonces, si es mejor que la incumbente, actualizamos el valor de la solución.

Por otra parte, en el caso recursivo, a la hora de definir los vecinos de un nodo de la exploración, establecemos como condición de factibilidad que el nuevo vértice no haya sido visto antes.

En la Figura 7.12 hay el mismo ejemplo sencillo que se utiliza en el código que va con el libro en este problema. Es una instancia pequeña que tan solo pretende poder ser depurada para la comprensión de todo ello. El grafo de entrada es el mismo que el de otros capítulos.

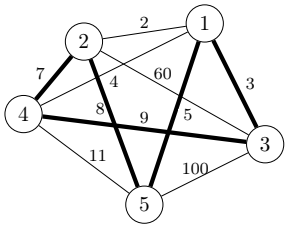


Figura 7.12: *Instancia solucionada del problema del viajante con valor* $z^* = 32$.

De la eficiencia de este último procedimiento, no vamos hacer nincún comentario. Abrimos n caminos y cada un de ellos abre otros $n-1$ que abrirán $n-2$. Debe andar por $\Omega(n!)$, lo cuál enuciamos con la boca pequeña.

7.3.5 Modelos Poliédricos

Un modelo poliédrico es sistema de inecuaciones lineales. En esta sección se habla de programación matemática. En concreto, de la programación lineal, ya que las aproximaciones más serias al TSP se han obtenido a partir de metodologías de esta disciplina. Dentro de ella, el ámbito desde el que se enfrenta el TSP que mejores soluciones ha proporcionado es el de los problemas de enrutamiento por nodos.

No sólo en la progrmación lineal, sino en toda la investigación operativa, las aplicaciones utilizan *solvers*, solucionadores de sistemas de ecuaciones, que se pronuncia con acento en la o, sólver. Entre los más conocidos hay el CPLEX, adquirido no hace mucho por IBM. El sólver CPLEX es la referencia oficial de la comunidad científica para medir tiempos de CPUs en trabajos de investigación.

También el sólver GLPK, de código abierto, merece ser mencionado. A cualquiera de ellos se le puede pedir que resuelva un problema de programación lineal, o problema lineal (PL), e incluso un problema de programación lineal entera, o problema lineal entero (PLE). La diferencia entre un PL y un PLE es que un PL admite valores continuos para las variables en las soluciones factibles, y en cambio un PLE tan solo admite valores enteros. Se entiende así que un PL pueda ser utilizado como relajación de un PLE, y por tanto sirva para obtener cotas. Eso es como suponer que el viajente puede pasar media vez por una carretera. De ser, es imposible, pero numéricamente, tener esa posibilidad le dará valores útiles de cotas.

El solo hecho de que se pretenda encontrar soluciones enteras, ya hace que el problema sea $\Omega(2^n)$, ya que aunque que al sólver se le pueda pedir soluciones enteras, lo que se le está pidiendo es que sea él quien realice la exploración por ramificación y poda. Y por tanto, será él quien tardará. Es decir, para conseguir tiempos polinómicos hay que trabajar con variables continuas. Entonces, cuando la solución obtenida tiene valores fraccionales como $x_i = 1/2$ para alguna $i \in \{1, \ldots, n\}$, la única manera de eliminarlos es creando dos nuevas instancias, una añadiendo la restricción $x_i \leq 0$ y la otra con $x_i \geq 1$. La solución del problema original será la mejor solución entre las dos subinstancias.

Llegados a este punto, resulta indispensable mencionar el Método Símplex como el algoritmo más utilizado en la teoría de optimización. Este algoritmo tan importante, fue introducido por el norteamericano George Bernard Danzig en 1947, sacudiendo el conocimiento algorítmico. Es el que utilizan los sólveres para resolver los sistemas de inecuaciones lineales. A grandes rasgos, el símplex se basa en un teorema en el que se postula que para cualquier problema lineal, siempre existe algún vértice del poliedro de factibilidad donde se produce el valor óptimo de la función objetivo. El algoritmo símplex se mueve de vértice en vértice agilizando la búsqueda del óptimo de forma espectacular. La bibliografía sobre este método es extensa, desde lo más sencillo, [19], hasta estudios más especializados, [11, 4], o también más pragmáticos, [8].

Respeto a la eficiencia, podemos entender que el símplex se utiliza una única vez para resolver un PL, y un número exponencial de veces, respeto al número de variables de la instancia, para resolver un PLE. No obstante, curiosidades de la vida, el símplex no es un algoritmo polinómico en el caso peor, aunque casi nunca se produce... y siempre se trabaja considerando el símplex como un algoritmo polinómico.

En definitiva, para resolver un problema de optimización combinatoria, la programación matemática se esfuerza en describir los poliedros que caracterizan las regiones de factibilidad de cada problema. En otras palabras, a definir en un sistema de inecuaciones lineales las propiedades que deben cumplir las soluciones de los problemas. Como se verá a continuación, para el TSP se definen las variables binarias, o sea enteras no inferiores a cero ni superiores a uno. Eso restringe la región de factibilidad del problema a los vectores $x \in \{0, 1\}^{|E|}$. Cuando un poliedro es acotado, entonces se denomina *polítopo*. Para los problemas de rutas en que se permite pasar varias veces por una misma arista,

tenemos poliedros. Para el TSP, es un polítopo.

El Polítopo del TSP

Hagamos una breve incursión en la descripción del polítopo del TSP.

Denotamos por $V = V(\mathcal{K}_n)$ y $E = E(\mathcal{K}_n)$ los conjuntos de vértices y de aristas del grafo completo de n nodos, $\mathcal{K}_n$. Ergo, la función de costes está definida sobre E.

En esta sección se utiliza la notación compacta, para las funciones reales definidas sobre conjuntos de aristas. Dado un subconjunto $A \subseteq E$, y una función real, $f : E \to \mathbb{R}$ que a cada $e \in E$ le asocia un valor $f_e \in \mathbb{R}$, podemos denotar por $f(A)$ la suma de los valores f_e, para todas las aristas $e \in A$. O sea,

$$f(A) = \sum_{e \in A} f_e, \qquad A \subseteq E. \tag{7.1}$$

Formular un PLE, o problema lineal entero, implica, antes que nada, definir las variables que intervienen, cosa que está relacionada con la dimensión del poliedro. En la formulación que aquí se desarrolla, se utilizan $m = |E| = n(n-1)/2$ variables. Eso es, una variable binaria por arista que nos indicará si la arista pertenece o no a la solución. Una ruta se caracteriza así con un vector $x \in \mathbb{Z}^m$, donde cada x_e, para $e = 1, \ldots, m$ representa si la arista e forma parte de la solución, o no.

En cualquier caso, tal como se ha dicho, por el mero hecho de definirlas enteras, el algoritmo ya se va más allá de $\Omega(2^n)$. Por tanto, empezaremos relajando esta condición. Ignoramos que las variables deban ser binarias, o sea enteras, permitiendo que del sólver podamos obtener soluciones de valores continuos. Ahora, pues, nos dedicamos a buscar una cota inferior para la solución. Aún así, si para alguna instancia concreta tuviéramos la suerte de que la solución del LP fuera entera, podríamos estar seguros que es una solución óptima para el PLE. Para entender esto, pensad que si la persona más alta de una aula de la universidad es una chica, entonces seguro que la chica más alta es la misma persona. El conjunto de las personas representa la relajación del conjunto de las chicas.

Seguidamente, hay que formalizar la función objetivo. Queremos minimizar el coste de la ruta completa. Como el coste de cada arista viene dado por la instancia del problema, $c \in \mathbb{R}^m$, la función objetivo es simple.

$$\min \sum_{e=1}^{m} c_e x_e \tag{7.2}$$

O sea, que el problema es encontrar el vector $x \in \{0,1\}^m$ que minimice la

expresión 7.2. No obstante, no nos vale cualquier vector x de ese hipercubo, sinó tan solo aquellos que representen rutas en el grafo $\mathcal{K}_n$. La única cosa que queda por hacer, por tanto, es caracterizar qué vectores de ese espacio representan rutas. Nos disponemos pues a coleccionar planos en el espacio $\{0,1\}^m$, que limiten el conjunto de soluciones factibles. Vamos a por ello.

Por un lado, sabemos que cualquier vértice de la ruta ha de tener exactamente dos aristas incidentes. Estas restricciones acostumbran a denominarse *restricciones de paridad*. Utilizando la notación $e = uv \in E$ para las aristas del grafo $\mathcal{K}_n$, o sea, haciendo referencia a sus dos vértices, se puede expresar esta condición con n ecuaciones, una por cada vértice $v \in V$.

$$\sum_{u=1}^{n} x_{uv} = 2, \qquad v \in V \tag{7.3}$$

Obtenemos una aproximación inicial al modelo que se está formulando agrupando las expresiones de la función objetivo (7.2), y las ecuaciones (7.3).

$$\begin{aligned} \text{(TSP)} \qquad \text{minimizar} \quad & \sum_{e=1}^{m} c_e x_e \\ \text{satisfaciendo} \quad & \sum_{u=1}^{n} x_{uv} = 2, \qquad v \in \{1,\ldots,n\} \\ & x \in \{0,1\}^m \end{aligned} \tag{7.4}$$

Y hasta aquí las cosas están bien claras. No hemos llegado muy lejos. Si alimentáramos un sólver con un modelo como el de la formulación (7.4), seguramente obtendríamos soluciones disconexas. Uns cuantos ciclos desperdigados por aquí y por allá de manera que efectivamente todos los vértices tendrían grado 2, Pero la ruta representada por el vector x que nos retornase el sólver no nos valdría. Es necesario refinar el espacio de factibilidad con alguna cosa más que lo que se impone con las restricciones de paridad.

Hay unas desigualdades que se llaman *restricciones de conectividad* y que evitan que la solución forme ciclos disconexos. Eso es, hace que las soluciones sean conexas. Para poder expresarlas es conveniente definir el conjunto de aristas interiores a un subconjunto vértices.

Dado un subconjunto de vértices $S \subset V$, las aristas interiores de S que denotamos por $E(S)$, son las que tienen ambos vértices dentro de S.

$$E(S) = \{uv \in E(\mathcal{K}_n) \mid u, v \in S \subset V\}.$$

Las siguientes desigualdades fueron introducidas el 1954 por Danzig, Fulkerson y Johnson, [10]. De todas formas, no son difíciles de inventar. Lo que expresan forma parte del sentido común.

$$x(E(S)) \leq |S| - 1, \qquad S \subset \{1, \ldots, n\}. \tag{7.5}$$

O sea, en un subconjunto de 3 nodos, como mucho puede haber dos aristas interiores. Claro.

Con las expresiones (7.3) y (7.5) el poliedro del TSP queda totalmente descrito. Eso significa que cualquier vector $x \in \{0,1\}^m$ que cumpla estas condiciones es una solución factible.

Aun así, el número de desigualdades del tipo (7.5) es muy grande. Concretamente $2^n - 2$, ya que para cualquier subconjunto posible $S \subset V$, hay una desigualdad. En la Sección 6.5.1 queda claro que el número de subconjuntos posibles que se puede hacer con un conjunto de n elementos es 2^n. De éstos, quitamos los subconjuntos $S = \emptyset$ y el $S = V$. Todo ello hace pensar que el problema lineal que se está planteando, a pesar de ser un LP y no un PLE, o sea, a pesar de haber relajado las condiciones de integridad para las variables, no será polinómico. Es decir, no se podrá resolver con un algoritmo polinómico.

Técnicas de Planos Secantes

Ante tal cantidad de desigualdades actúan las técnicas de planos secantes. Son algoritmos iterativos que en su inicialización construyen un modelo inicial. En este modelo se limitan a poner tan solo unas cuantas restricciones. Por ejemplo, las que $|S| \leq 3$. En la misma fase de inicialización del procedimiento se llama al sólver una sola vez, con este modelo reducido. Entonces se entra en un bucle en el que en cada iteración se recoge la solución del último problema solucionado, y se analiza para encontrar cuál de las desigualdades no añadidas todavía, no se está satisfaciendo por esta solución. O sea, qué desigualdad está violada por la solución actual, y que por tanto hay que añadir al modelo, y reenviarlo al sólver. Esta parte se conoce como el problema de separación. Del conocimiento de métodos de separación para desigualdades violadas depende esencialmente la polinomicidad del PLE. Si tenemos métodos polinómicos de separación para las desigualdades no introducidas ni en el modelo inicial ni en las separaciones posteriores, el problema lineal se resuelve en tiempos polinómicos. Cuando no conocemos estos procedimientos, o los conocemos pero no son polinómicos, tenemos un problema no resuelto.

Mientras se encuentre alguna desigualdad violada, se añade al sistema de inecuaciones y se reenvia a solucionar. Y así hasta que ya no se encuentren más desigualdades violadas. Entonces puede ser que la solución sea entera, afortunadamente.

Seguidamente se muestra la estructura típica de funcionamiento de los algoritmos de planos secantes.

```
tecnica planos_secantes()
{
    describir_modelo_inicial(problema)
    solucion ← solucionar(problema)
    restriccion ← separacion(solucion)
    mientras (violada(restriccion)) {
        añadir_restriccion_al_modelo(restriccion)
        solucion ← solucionar(problema)
        restriccion ← separacion(solucion)
    }
    si (entera(solucion)) optimo = valor(solucion)
    sino optimo = branch&cut(solucion);
}
```

Si en ninguna iteración se consigue una solución entera y no se saben separar nuevas desigualdades violadas es porque no tenemos bien resuelto el problema de la separación. O peor aún, no tenemos caracterizado el poliedro del problema, lo cuál significa que ni siquiera sabemos qué forma tienen las desigualdades que, si las coneciéramos, todavía sería necesario aprender a separarlas...

En muchos casos los problemas de separación son más complicados que los de las instancias iniciales que pretenden resolver. Son casos en los que claramente el remedio es peor que la enfermedad. Cuando eso ocurre, no habrá otra alternativa que recorrer a la ramificación y poda, desafortunadamente. Si podemos hacer uso de heurísticas para las cotas superiores, mejor.

En este capítulo se ha presentado la técnica algorítmica aplicada con la que se han obtenido los mejores resultados en una amplísima gama de problemas. Los algoritmos enumerativos sitúan los planteamientos de los problemas de optimización con variables enteras en las antípodas de donde los sitúan las demás metodologías, ya que parten de la idea de la enumerabilidad de las soluciones factibles para explorarlas todas. El inconveniente más importante que tiene la ramificación y poda es que sus eficiencias se van más allá, con holgura, de los tiempos polinómicos. Llegados a este punto, la ramificación y poda se centra en establecer mecanismos para dejar de explorar las combinaciones de valores de las variables enteras que considera no prometedoras. Para decidir que una combinación de valores no es prometedora se utilizan cotas del valor de la solución del problema. Para calcular estas cotas se usan soluciones heurísticas obtenidas con algoritmos voraces por una parte, y relajaciones del problema por otra.

Al final del capítulo, se ha realizado una breve incursión en la programación matemática en la qué se han ilustrado el método simplex, los modelos poliédricos, y los algoritmos de planos secantes.

Capítulo 8

Complejidad Algorítmica

Cuidado con lo que deseas, porque quizás lo consigas. Es innegable que el tamaño n de los datos de los problemas y los tiempos de los algoritmos, aquellos conceptos instaurados en las primeras páginas de esta cinta, nos han sido de gran utilidad. Anhelábamos disponer de capacidad expresiva para la eficiencia de los algoritmos, y quedamos satisfechos. Hemos conseguido lo que nos habíamos propuesto. A pesar de todo, tenemos suerte. Nos quedan cabos por atar. De todo lo dicho hasta ahora, hay alguna cosa que nos inquieta. Se trata de estos últimos problemas del Capítulo 7. Teniendo en cuenta las eficiencias impresentables con las que acabábamos, no se puede decir que los hayamos resuelto. Es inadmisible que el tiempo para resolver un problema se multiplique a incrementos lineales del tiempo de plantearlo. Alguna cosa debemos de decir cuando, después de aprender a medir la dificultad de resolver los problemas, decimos que los hay que no sabemos resolver. Porque, seamos francos, un problema que se resuelve con una eficiencia de $\Omega(2^n)$, no se puede decir que se haya resuelto. ¿O sí?. Dejar una cosa para mañana y no hacerla es casi lo mismo.

Con ganas de seguir, habitamos el universo no polinómico donde los procedimientos ya no se comportan como sería deseable. En este capítulo intentaremos al menos, conocerlos. Es un capítulo que no sirve para resolver nuevos problemas. Sirve para darse cuenta de que no sabemos resolver los difíciles. Se trata de caracterizarlos para poderlos identificar.

Bien, cuando en el inicio no sabíamos como abrazar todas las posibles entradas, hemos establecido el concepto de tamaño de los datos. Distinguiendo entre casos medios y peores cuando eran notables, hemos formulado una teoría que, como si dijéramos, considera todas las entradas posibles para los algoritmos. Finalmente hemos topado contra la evidencia de que había problemas complejos. Problemas que no tenemos forma de resolver en tiempos satisfactorios. La salida más digna que nos queda es bajar la cabeza, y procurar conocer cuanto más mejor aquello que no sabemos resolver.

Este capítulo se estrena con la distinción entre los problemas computacionales y los decisionales. Se ejemplifican algunas versiones decisionales de algunos problemas concretos, y seguidamente se pasa a la definición de algoritmo polinómico. Esto permitirá definir las súper famosas clases de problemas $\mathcal{P}$ y $\mathcal{NP}$. A partir de aquí, se podrá comprender el concepto de reducción polinómica, y definir el conjunto de los problemas díficiles, $\mathcal{NP}$-duros. Comenzaremos a conocer los problemas con mayúsculas. Primero de todos, SAT, el problema de la satisfactibilidad de expresiones lógicas. En un acto de fe, asumiremos el Teorema de Cook, que clasifica SAT como $\mathcal{NP}$-completo. El capítulo cierra con una serie de problemas $\mathcal{NP}$-duros que se demuestra que son $\mathcal{NP}$-completos mediante reducciones polinómicas.

8.1 Problemas Computacionales y Decisionales

Para desenmarañar aquello que queremos comprender, empezamos definiéndolo. Seguidamente se establece una definición de problema computacional como relación entre conjuntos. Los problemas computacionales son los que se han estado viendo a lo largo de todo el libro, aquí no se hace más que formalizarlos.

El producto cartesiano de dos conjuntos es el conjunto de todas las parejas posibles con un elemento de cada conjunto.

Definición 8.1 Problema Computacional. *Dados un conjunto de entradas E, y un conjunto de salidas S, un problema computacional R es un subconjunto del producto cartesiano de los dos conjuntos, $R \subseteq E \times S$, que relaciona cada entrada $x \in E$ con su salida $y \in S$.*

Para clasificar estos problemas según sus conjuntos de entrada se ha hecho lo que se ha podido. El fruto de ese esfuerzo ha sido poder añadir a la definición que además, en todos los problemas computacionales hay definida una función de tamaño, $|x| : E \to \mathbb{N}$, que a cada entrada posible $x \in E$ le hace corresponder un número natural $n = |x|$.

Ahora, pues, miremos si de clasificarlos a partir de sus salidas podemos sacar algún provecho. Para empezar, reducimos nuestras aspiraciones y nos centraremos sólo en aquellos problemas computacionales que además, son también problemas decisionales.

Definición 8.2 Problema Decisional. *Dados un conjunto de entradas E, y el conjunto de salidas $\{0,1\}$, un problema decisional R es un subconjunto del producto cartesiano de los dos conjuntos, $R \subseteq E \times \{0,1\}$, que relaciona cada entrada $x \in E$ con su salida $y \in \{0,1\}$.*

Está bien claro que la Definición 8.2 coincide con la que ha sido utilizada

en los últimos capítulos. Bien, de momento aquí, hasta ahora tan solo se ha añadido una formalización.

Profundicemos en la cosa. De alguna manera se puede interpretar que cuando nos llega cualquier información, el hecho de creerla o no, es un problema decisional. Se entiende que los problemas decisionales se resuelven a cierto o falso. Y por tanto, para cualquier problema decisional se puede establecer una partición en el conjunto de entrada. Se pueden distinguir dos subconjuntos de E, que lo particionan.

$$L = \{x \in E \mid R(x, 1)\}$$
$$E \setminus L = \{x \in E \mid R(x, 0)\}$$

En palabras, L es el conjunto de las instancias posibles de entrada al problema R tales que sucede $R(x, 1)$. O sea, el conjunto de las verdades. E es todo lo que se puede decir, L es lo que es verdad.

A los $x \in L$, las denominaremos instancias positivas. En cambio, a las otras, las que $x \in E \setminus L$ no les llamaremos instancias negativas. Hay que ir con mucho cuidado con las palabras que se utilizan. Nos estamos moviendo en un terreno resbaladizo.

El conjunto de instancias positivas caracteriza un problema decisional. Denotaremos por $L \subseteq E$ un problema decisional. Para verbalizarlo de una manera ágil, diremos *sea L en E un problema decisional.*

8.1.1 Versiones Decisionales de Problemas Computacionales

Un problema de optimización, por ejemplo de maximización, tiene siempre una versión decisional asociada. Esta versión del problema, se soporta en una variable, que normalmente se llama k, para transformar la pregunta de, ¿Cuál es el valor máximo?, por la pregunta, ¿Es mayor que k?.

Es muy sencillo. Seguidamente se dan las versiones decisionales de los problemas de optimización tratados en alguna parte del libro.

- *Versión decisional del problema de la mochila,* Sección 5.3. Dados n objetos que tienen pesos w_i para $i = \{1, \ldots, n\}$ y valores v_i, para $i = \{1, \ldots, n\}$, y un valor k, ¿Es posible conseguir un valor mayor que k, siempre y cuando el peso total no supere la capacidad de peso W de la mochila?.
- *Versión decisional del problema de caminos mínimos*, Sección 5.5. Dados un grafo, $G(V, E)$, dos vértices, $s, t \in V$, una función real de distancias asociada a las aristas, $d : E \to \mathbb{R}$, y un valor k, ¿Existe algún camino entre los dos vértices s y t con una distancia inferior a k?.

- *Versión decisional del problema del árbol de expansión mínima*, Sección 5.6. Siendo $G(V, E)$ un grafo conexo no dirigido, c una función real de costes no negativa asociada a las aristas, $c : E \rightarrow \mathbb{R}^+$, y un cierto valor k, la versión decisional del problema del árbol de expansión mínima consiste en afirmar o negar que exista un subconjunto de aristas $T \subseteq E$, de manera que la suma de los costes de todas ellas sea menor a k, y que incida en todos los nodos del conjunto V.

- *Versión decisional del problema de la asignación,* Sección 7.3.3. Dados n proyectos, n empresas, la matriz de costes $n \times n$ con lo que cobra cada empresa por realizar cada proyecto, y un cierto valor k, ¿Es posible asignar un proyecto a cada empresa con un coste total inferior a k?.

8.2 Algoritmos Polinómicos

La definición dada para algoritmo sí que viene de lejos. Ya en el Capítulo 1 se postulaba como sigue.

Definición 8.3 Algoritmo. *Un algoritmo A es un procedimiento para resolver problemas de manera que transforma unos datos x, de un conjunto de posibles datos de entrada E, en una información y, de un conjunto de posibles salidas S, obtenida a partir de ellos.*

$$A : \underset{x}{E} \rightarrow \underset{y}{S}.$$

Y también del mismo capítulo, Sección 1.2, importamos las definiciones de tamaño de los datos como una función que asocia un número natural a cada posible entrada de un algoritmo, que en términos formales venía a ser

$$|\ .\ | : \underset{x}{E} \rightarrow \underset{n=|x|}{\mathbb{N}},$$

La definición que en su momento se dio para el tiempo de un algoritmo para una entrada de tamaño n, $T_A(n)$, y que sintetizado en un solo número para cada n es el del caso peor, también vamos a utilizarla ahora

$$t_A(n) = \underset{x \in E}{max} \{T_A(n) \ \mid \ |x| = n\}.$$

Pues bien, se utilizan todas estas definiciones para definir lo que era previsible.

Se ha estado hablando durante 317 páginas, y todavía no se había formalizado con el rigor que se está haciendo en este momento. Ha llegado la hora de hacerlo.

Lo que sigue no es más que una formalización más.

Definición 8.4 Algoritmo Polinómico.

$$A \text{ es un algoritmo polinómico} \Leftrightarrow \exists k \geq 0 \mid t_A(n) \in \Theta(n^k).$$

8.3 $\mathcal{P}$ y $\mathcal{NP}$

Se debe comprender que se está haciendo acopio de conceptos par poder caracterizar, en última instancia, los problemas difíciles. Desde el inicio de este capítulo se tiene asumido que hay problemas que no los sabemos resolver en tiempo polinómico. Nuestra intención estriba en reconocer estos problemas. Ser capaces de decir, que efectivamente no lo podemos resolver en tiempo polinómico y demostrarlo diciendo que este problema es tan difícil o más que algún otro que ya sepamos que es difícil. Esa es la idea.

Sigamos pues, equipándonos de definiciones.

Definición 8.5 Clase de complejidad $\mathcal{P}$. *Conjunto de problemas decisionales decidibles en tiempo polinómico.*

$$L \subseteq E \in \mathcal{P} \Leftrightarrow$$
$$\exists \text{ un algoritmo polinómico } A : E \to \{0,1\} \mid$$
$$x \in L \Leftrightarrow A(x) = 1.$$

Un problema decisional $L \subseteq E$ pertenece a la clase $\mathcal{P}$ si se puede implementar un algoritmo polinómico que, frente a una instancia cualquiera del problema, si la solución es 1, la salida del algoritmo sea 1. Y si la solución del problema es 0, entonces la salida del algoritmo también sea 0.

Seguidamente se verá la definición de la clase $\mathcal{NP}$. Aun así, para contrastarla con la clase $\mathcal{P}$ y a riesgo de hablar de un concepto antes de definirlo, se enuncia primero una propiedad. Así pues, antes de definirla, de la clase $\mathcal{NP}$ podemos apuntar la siguiente

Proposición 8.1 Sobre la clase de complejidad $\mathcal{NP}$. *Para el conjunto de problemas decisionales decidibles en tiempo polinómico indeterminista, sucede que*

$$L \subseteq E \in \mathcal{NP} \Rightarrow$$
$$\exists \text{ un algoritmo polinómico } A : E \to \{0,1\} \mid$$
$$x \in L \Rightarrow A(x) = 1.$$

Si un problema decisional $L \subseteq E$ pertenece a la clase $\mathcal{NP}$ entonces se puede implementar un algoritmo polinómico que, para cualquier instancia positiva del

problema, o sea que la solución es 1, la salida en tiempo polinómico del algoritmo sea 1. Y si la solución del problema es 0, entonces la salida del algoritmo no se sabe.

La Proposición 8.1 no es una definición, ya que no nos dice nada de como se comporta el algoritmo frente instancias de certeza desconocida. Por eso, hay que adentrarse en el concepto, de manera que a partir de la definición se pueda discriminar sin duda alguna qué problemas pertenecen a esta clase y cuáles no. Se establece la siguiente

Definición 8.6 Clase de complejidad $\mathcal{NP}$. *Conjunto de problemas decisionales decidibles en tiempo polinómico indeterminista.*

$$
\begin{aligned}
&L \subseteq E \in \mathcal{NP} \Leftrightarrow \\
&\quad \exists \textit{ un conjunto } E', \\
&\quad y \ \exists \textit{ un algoritmo polinómico } B : E \times E' \to \{0,1\} \mid \\
&\qquad\qquad\qquad\qquad x \in L \Leftrightarrow \exists y \in E' \mid B(x,y) = 1.
\end{aligned}
$$

Apuntes esclarecedores de la Definición 8.6 son,

- $L \subseteq E$ es el problema decisional perteneciente a la clase $\mathcal{NP}$, que se está definiendo.
- E' es un conjunto cualquiera. El que convenga. Sin embargo, en el fondo, serán las soluciones de los problemas x. Poco a poco.
- B es un algoritmo que se denomina *verificador polinómico*.
- x es una instancia del problema decisional $L \subseteq E$.
- y es una solución del problema x. Se le llama *demostración*, *testimonio*, *certificado*,...

Después de las dos definiciones anteriores, respiremos hondo. Clasificamos los problemas decisionales según como se comporten ante instancias de las que sabemos la solución. Es extraño. Bien, lo primero que hay que hacer para digerir estas definiciones, son algunas consideraciones que como mínimo, ayudarán a familiarizarse con los conceptos.

La manera más concisa de caracterizar $\mathcal{NP}$ es con el sintagma *respuestas polinómicas a instancias positivas*. Eso es $\mathcal{NP}$, respuestas polinómicas a instancias positivas.

Otra manera de entenderlo es que un problema pertenece a $\mathcal{NP}$ si cuando tenemos una solución podemos comprobar, en un tiempo polinómico, que efectivamente es una solución.

Por ejemplo, la versión decisional del problema del viajante es un problema $\mathcal{NP}$. El enunciado de esta versión decisional sería,

A partir del grafo completo de n vértices $\mathcal{K}_n$, de una función real de costes $c : E(\mathcal{K}_n) \to \mathbb{R}$, y dado un valor k, ¿Existe algún ciclo simple que recorra los n vértices con un coste menor que k?.

Se ha visto en el Capítulo 7 que resolver este problema es $\Omega(2^n)$. Pero en cambio, un algoritmo que dado el grafo completo $\mathcal{K}_n$, la función de costes c, y una solución al problema del viajante $X \subset E(\mathcal{K}_n)$ de coste $c(X) < k$, fuese capaz de verificar que la solución dada realmente cumple las condiciones de factibilidad y además cuesta menos que k, efectivamente podría ser un algoritmo polinómico. Es como decir, mira, aquí tienes el problema y aquí la solución, ¿Lo es o no?. Y comprobarlo, ya se ve que podría hacerse en $\Theta(E)$. Sería polinómico. Por esta razón al elemento X se le llama demostración, certificado o testimonio.

La palabra indeterminista, mejor que no la utilicemos demasiado, porque es complicada. Lo que sí que podemos decir es que un problema decisional es $\mathcal{P}$ si cuando se le dice alguna cosa, el algoritmo que lo resuelve replica en tiempo razonable si eso es cierto o falso. En cambio, si es $\mathcal{NP}$, entonces cuando se le dice una verdad, responde que es cierto, pero cuando se le dice una mentira, no responde. De manera que nunca sabremos si eso que se le ha dicho al fin y al cabo es mentira, o hay que seguir esperando porque tal vez es verdad.

Por un lado está bien claro que $\mathcal{P} \subseteq \mathcal{NP}$. Eso es importante y es preciso recordarlo. La $\mathcal{N}$ de $\mathcal{NP}$ no significa *No*. Significa *no-determinista*. Cometer el error de pensar que $\mathcal{P}$ y $\mathcal{NP}$ son contrarios es muy grave.

Que $\mathcal{P} \subseteq \mathcal{NP}$ es una consecuencia directa de la definición de ambas clases. Ahora bien, la conjetura de que $\mathcal{P} \neq \mathcal{NP}$, y por tanto $\mathcal{P} \subset \mathcal{NP}$, es una de las más vivas en la comunidad científica hoy día. Demostrar que efectivamente $\mathcal{P} \neq \mathcal{NP}$ tal como se conjetura, es uno de los temas abiertos que más espectación arrastra. Una pregunta del milenio. En definitiva, la cuestión es si se pueden resolver problemas difíciles con métodos fáciles. Conjeturamos que no. Que los humanos no som cortos. Que hay problemas objetivamente más complejos que otros.

8.4 Reducciones Polinómicas

Una reducción polinómica es un algoritmo polinómico, entendido en el sentido más amplio, un método para transformar instancias de un problema en instancias de otro.

Definición 8.7 Reducción Polinómica. *Dados los problemas decisionales $L \subseteq E$ y $L' \subseteq E'$, decimos que un algoritmo polinómico $R : E \to E'$ es una reducción*

polinómica de $L \subseteq E$ a $L' \subseteq E'$ si cuando $x \in L \Rightarrow R(x) \in L'$, y a la inversa, que cuando $R(x) \in L' \Rightarrow x \in L$.

Una reducción polinómica es un programa para transformar instancias, enunciados, datos de entrada. No hace falta pensar en programa como algoritmo, sinó como manera, que tiene connotaciones más sencillas y al fin y al cabo, es lo mismo. Cuando existe una manera R de transformar instancias con las características de la Definición 8.7, entonces decimos que L en E es reducible en tiempo polinómico a L' en E'.

Les reducciones polinómicas no son commutativas. Por este motivo se denominan *reducciones* significando cierta asimetría. Que un problema se pueda reducir a otro no quiere decir que el otro se pueda reducir al uno. Esto es importante porque pseudo ordena la complejidad de los problemas. En otras palabras, si un problema se puede reducir a otro, quiere decir que si tuviéramos una manera de solucionar ese otro podríamos solucionar también el primero. O sea, que ese otro es tanto o más difícil que el primero. Conviene insistir en este razonamiento. No se trata de aprendérselo de memoria. Es fácil deducirlo cada vez que se requiera.

Si el problema α se reduce polinómicamente al problema β, entonces β es tanto o más difícil que α, ya que si se dispusiera de un algoritmo para solucionar β, automáticamente se tendría también un algoritmo para solucionar α.

En síntesis, si se sabe hacer lo difícil, entonces se sabe hacer lo fácil. Cuando se utilizan reducciones polinómicas para demostrar complejidades, el problema reducido es el que queremos demostrar que es difícil, o sea, es el problema del que queremos demostrar que su complejidad es tal. Y decimos, "si supiéramos resolver el nuestro, entonces podríamos resolver aquel otro, que ya se sabe que no lo sabemos resolver".

Dentro de la comunidad científica hay mucha confusión respeto a ese extremo. Hay tanta, que a menudo se habla de transformaciones polinómicas en lugar de reducciones para evitar caer en el error de qué se reduce a qué. Y no es difícil!

Por ejemplo, el Problema del Cartero Chino, *the Chinese Postman Problem*, o directamente el CPP, es el problema de cruzar todas las aristas de un grafo con pesos, minimizando el coste del ciclo solución. En la Figura 8.1*(a)* se muestra una instancia y en la Figura 8.1*(b)* la única solución óptima con valor $z^* = 83$. Esta solución repite la arista de coste 9.

En cambio, el Problema del Cartero Rural, *the Rural Postman Problem*, o directamente el RPP, es el problema de cruzar algunas aristas, llamadas *requeridas*, de un grafo con pesos, minimizando el coste del ciclo solución. En la Figura 8.2*(a)* se muestra una instancia con las aristas requeridas en negrita. El conjunto de aristas requeridas de esta instancia tiene dos componentes conexas.

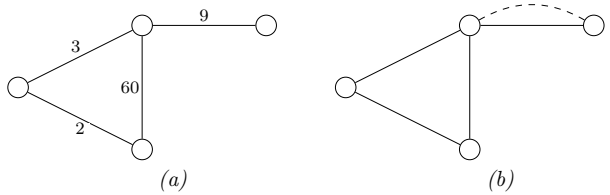


Figura 8.1: *(a) Instancia del CPP; (b) Única solución óptima, con* $z^* = 83$.

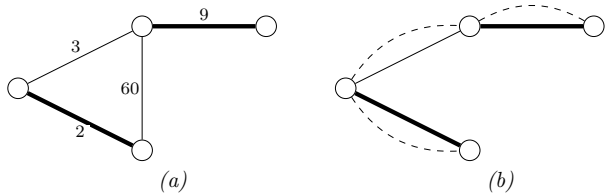


Figura 8.2: *(a) Instancia del RPP; (b) Única solución óptima, con* $z^* = 28$.

En la Figura 8.2*(b)* se puede ver la única solución óptima con $z^* = 28$ que repite tres aristas y no utiliza la de coste 60.

Es decir, el CPP es el caso particular del RPP en el que todas las aristas son requeridas. Por tanto, está bien claro que si supiéramos resolver el RPP, entonces sabríamos resolver el CPP. Una reducción polinómica del CPP al RPP es un algoritmo que transforme instancias del CPP a instancias del RPP, cosa que ya se ve que es bien fácil. Sólo se trata de decir que todas las aristas del CPP son requeridas para el RPP.

Aun así, la frontera entre la clase $\mathcal{P}$ y la $\mathcal{NP}$ cae en medio de los dos problemas. El CPP $\in \mathcal{P}$, es decir, que se sabe resolver en tiempos acotados polinómicamente con el tamaño del grafo. El RPP $\in \mathcal{NP}$, es decir, que no se sabe resolver tan de prisa. A pesar de todo, o sea, a pesar de que sepamos resolver el CPP y no el RPP, lo que se ha dicho es cierto. Si tuviéramos un algoritmo para resolver en tiempo polinómico el RPP, podríamos resolver en tiempo polinómico el CPP. Que el CPP se pueda resolver de otras formas no tiene nada que ver.

8.5 Problemas $\mathcal{NP}$-duros y $\mathcal{NP}$-completos

Se dice que un problema es $\mathcal{NP}$-duro si cualquier problema $\mathcal{NP}$ se puede reducir a él. Es decir, es tanto o más difícil que cualquier problema $\mathcal{NP}$.

Los problemas $\mathcal{NP}$-duros son algunos de los problemas de $\mathcal{NP}$ que no están en $\mathcal{P}$. Aunque parezca presuntuoso, tenemos aquí una definición analítica, y por

tanto pretendidamente objetiva, de *difícil*. Los problemas $\mathcal{NP}$-duros también se llaman $\mathcal{NP}$-difíciles. En inglés es siempre $\mathcal{NP}$-hard. En cualquier caso, los problemas pertenecientes a $\mathcal{P}$ nunca se llaman *fáciles*. Todo merece respeto. A los problemas $\mathcal{P}$, los podemos seguir llamando *polinómicos*.

Y ahora agarraos.

Un problema es $\mathcal{NP}$-completo, si es $\mathcal{NP}$-duro y además, pertenece a $\mathcal{NP}$.

Y volvemos a respirar hondo.

La manera más fácil de demostrar que un problema pertenece a $\mathcal{NP}$ es observando que la verificación de una solución es polinómica. La manera habitual, no fácil, de demostrar que un problema es $\mathcal{NP}$-duro es llegando a él a partir de una reducción originada en un problema que sabidamente es $\mathcal{NP}$.

Hay otras clases de complejidad a parte de las que se han visto en esta sección. Aquí no se profundizará más en esta materia. Aun así, conviene mencionar que normalmente se usa el concepto de $\mathcal{NP}$-duro cuando se habla de problemas de optimización. El RPP es un problema $\mathcal{NP}$-duro. También se tiene mucho en cuenta que un problema tenga un planteamiento decisional en el momento de calificarlo de $\mathcal{NP}$-completo.

Dejando aparte las formalizaciones teóricas, desde una óptica mucho más intuitiva, que un problema sea $\mathcal{NP}$-completo se puede interpretar de una manera que hace que sean un tipo de problemas muy cotidianos. Los problemas $\mathcal{NP}$-completos son una especie de problemas como "semidefinidos", o algo así. Problemas que cuando se resuelven, se resuelven, pero cuando no, nunca sabemos si podemos esperar una resolución, o no. Buscar. Probablemente, buscar sea el problema más antiguo de la humanidad. Y más si se tiene en cuenta que la navegación puede suponer perderse. Las personas, a lo largo de todos los tiempos, se han perdido yendo a los lugares. O más materialmente todavía. Perdemos las llaves de casa. Y entonces, ¿Qué hacemos?. Buscarlas. Las buscamos y las buscamos y puede ocurrir que las encontremos. Entonces ya está, el problema está solucionado. Pero cuando no, cuando no las encontramos, ¿Qué?.¿Cuándo se puede considerar solucionado haber perdido las llaves mientras no se encuentren?. Bien, llega un momento en que uno dice que basta. Ya conseguirá una copia de alguna manera, o llamará al cerrajero. Pero, ¿En qué momento?. Esta es una decisión realmente difícil. Si tuviéramos un diablillo que nos dijera que ya hay bastante. Que no hace falta seguir buscando porque no encontraremos las llaves..., sería fenomenal. A pesar de que igualmente deberíamos de hacer copias, sería de una utilidad enorme saber que ya no merece la pena buscar más. Pues bien, este diablillo es la y de la Definición 8.6. De hecho, sería la solución del problema de tener que buscar las llaves.

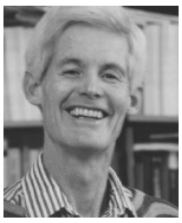
Hay otros ejemplos de problemas $\mathcal{NP}$-completos tanto o más cotidianos que el de buscar. La prevención también es un problema $\mathcal{NP}$-completo. Cuando cogemos el coche y nos ponemos el cinturón y corremos a una velocidad..., si

tenemos un accidente, el problema está resuelto. No hemos sido bastante previsores. Pero cuando no tenemos ningún accidente, entonces siempre podemos prevenirnos más. Y estaría muy bien tener un diablillo que nos dijera que ya está, que tranquilos que ya vamos lo bastante protegidos y no tendremos ningún accidente. Esto tiene mucha importancia, ya que mueve mucho dinero. Los grandes negocios de nuestro siglo se sustentan en problemas $\mathcal{NP}$-completos. Les casas de seguros viven de esto. El miedo es un problema $\mathcal{NP}$-completo.

No tiene nada de loco proclamar que por las rondas hay que ir a 80 Km/h. Incluso se podría decir, con tota la sensatez del mundo, que es una locura correr a más de 20 Km/h. O ¿Cómo se puede salir a la calle sin llevar casco?. ¿Cómo que sólo las motos?. Todo el mundo debería llevar casco, sólo faltaría.

En fin, siempre se puede ser más prudente, y más cuando se gana dinero en ello.

8.6 Teorema de Cook


Stephen Cook

Stephen Cook, (1939-...) es un ingeniero de computación norteamericano de Nueva York. En el artículo de 1971, "The complexity of Theorem Proving Procedures" estableció las clases de complejidad $\mathcal{P}$ y $\mathcal{NP}$, y por tanto Cook es el fundador de todo el mundo de la $\mathcal{NP}$-completitud. Esta teoría le valió el Premio Turing 1982, máximo reconocimiento en la disciplina de la computación. Su área principal de investigación ha sido la Complejidad Computacional. Sin embargo, también ha hecho incursiones en temas de semántica de lenguajes de programación y computación paralela. Actualmente es profesor de la Computer Science University of Toronto.

El Problema de Satisfactibilidad Booleana SAT

En esta sección se procura aislar, y dejar tan pelado como sea posible, un problema $\mathcal{NP}$.

Una instancia del Problema de Satisfactibilidad Booleana es una expresión booleana que combina variables booleanas utilizando operadores booleanos. La expresión es satisfactible si hay alguna asignación de valores de verdad a las variables que hace que la expresión resulte cierta. En la formulación (8.1) se puede ver un ejemplo.

$$f(x_1, x_2, x_3) = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3) \qquad (8.1)$$

Un poco de léxico respeto a la expresión (8.1). Les *variables* son x_1, x_2 y x_3. Se denominan *literales* a cada aparición de las variables, sea directa o negada. Para cada variable, pues, hay dos literales posibles. En el ejemplo, los literales son $x_1, \neg x_2, \neg x_3, \neg x_1, x_2$ y x_3. Negar una variable también se puede indicar con una barra sobre la variable, $\neg x \equiv \overline{x}$. Una expresión disyuntiva como $(x_1 \vee \neg x_2 \vee \neg x_3)$ es una *cláusula*. La expresión booleana f está en *forma normal conjuntiva* cuando se muestra como en la proposición (8.1), es decir, una conjunción de cláusulas disyuntivas.

Para enunciar el SAT desde un ángulo más plástico, se ilustra el problema CIRCUIT-SAT.

Definición 8.8 CIRCUIT-SAT. *Dadas n variables binarias $x_1, x_2, \ldots, x_n$ de entrada a un circuito lógico con una única variable de salida $f = f(x_1, x_2, \ldots, x_n)$, formado con puertas lógicas NOT, OR, AND, averiguar, si existe, una combinación de valores de las variables de entrada que activen la salida.*

En la Figura 8.3 se muestra el circuito correspondiente al problema de satisfactibilidad de la expresión (8.1)

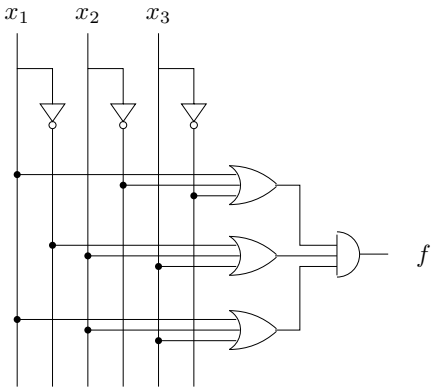


Figura 8.3: *CIRCUIT-SAT. Circuito correspondiente al problema* (8.1).

La manera más directa de resolver el problema planteado en la Figura 8.3 es por enumeración, con la tabla de verdad que se muestra en la Tabla 8.1.

Está claro pues, que al incrementar en una unidad la dimensión del problema, pasando de $n = 3$ a $n = 4$, el tiempo de resolución aumenta al doble.

Teorema 8.1 Teorema de Cook. *El problema de satisfactibilidad booleana SAT*

x_1	x_2	x_3	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Tabla 8.1: *Tabla de verdad del circuito de la Figura 8.3*

es $\mathcal{NP}$-completo.

A este teorema lo aceptamos sin entrar en los detalles de su demostración, que va más allá del ámbito que se muestra aquí. De todas maneras se observa que una de las dos condiciones para ser $\mathcal{NP}$-completo es fácil de demostrar. Si se dispone de una combinación para los valores de las variables que active la salida, comprobar que efectivamente es así requiere tan solo los tiempos necesarios para los cálculos de las puertas lógicas. Eso es claramente polinómico. Para la demostración de la otra condición, que todo problema de $\mathcal{NP}$ sea reducible a SAT, tan solo se menciona que en última instancia, cualquier programa informático puede ser sintetizado en un circuito como el de la Figura 8.3. Eso nos debería inclinar a pensar que efectivamente cualquier problema computacional se puede reducir a SAT. Para la demostración más rigurosa, se aconseja consultar el artículo "The complexity of Theorem Proving Procedures" al que se puede acceder a través de la wikipedia.

8.7 Algunas Reducciones

Un vez disponible un problema $\mathcal{NP}$-completo, ahora tan solo hay que obtener reducciones polinómicas para multitud de otros problemas, que rápidamente se podrá demostrar que también son $\mathcal{NP}$-completos.

8.7.1 Reducción de 3SAT a CONJUNTO INDEPENDIENTE

En lugar de reducir des de SAT, lo haremos desde 3SAT. El problema 3SAT es aquel caso particular de SAT en el cuál ninguna cláusula contiene más de tres literales. La reducción de SAT a 3SAT también forma parte de la contribución de Stephen Cook.

El problema del conjunto independiente consiste en encontrar el máximo número de vértices de un grafo tales que no haya ninguna arista que una dos de ellos. Aquí se plantea la versión decisional del problema del CONJUNTO INDEPENDIENTE.

Definición 8.9 CONJUNTO INDEPENDIENTE. *Dado un grafo $G(V, E)$ y un número natural k, determinar si G tiene un conjunto de k nodos independientes, es decir que no exista ninguna arista en E que conecte dos de ellos.*

En términos formales, el problema del CONJUNTO INDEPENDIENTE se puede enunciar como, $\exists A \subset V \mid |A| = k \wedge \forall u, v \in A, \{u, v\} \notin E$?

A pesar de que se hayan definido las reducciones como algoritmos polinómicos, en adelante se dan sólamente las líneas generales de cómo funcionaría el método sin entrar en detalles de bajo nivel.

Hacer la reducción de 3SAT a CONJUNTO INDEPENDIENTE significa partir de una instancia del problema 3SAT y describir las reglas para transformar esa instancia a una de CONJUNTO INDEPENDIENTE. Por ejemplo, se parte de la función

$$f = (a \vee \overline{b} \vee \overline{c}) \wedge (\overline{a} \vee b \vee \overline{c}) \wedge (\overline{a} \vee b \vee c) \tag{8.2}$$

Ahora se trata de plantear una instancia de CONJUNTO INDEPENDIENTE a partir del problema 3SAT de la fórmula (8.2). Para esto podemos coger la k que convenga. En la reducción, se supone que se parte de un 3SAT con m cláusulas. En el ejemplo, $m = 3$. Es preciso comprender que la reducción nos transforma una forma normal conjuntiva a un grafo. Se puede realizar como sigue.

- $k = m$. Es decir, dado un 3SAT con un cierto número de cláusulas, m, lo reduciremos a un CONJUNTO INDEPENDIENTE con número de vértices igual a $3 * m$.
- Crear el grafo del problema del CONJUNTO INDEPENDIENTE con

  $$V = \{(l, C_i) \mid l \text{ aparece en } C_i\}$$

  siendo l cualquier literal, y C_i cada una de las cláusulas. Los vértices de la instancia de CONJUNTO INDEPENDIENTE según el ejemplo de la fórmula (8.2) se muestran en la Figura 8.4

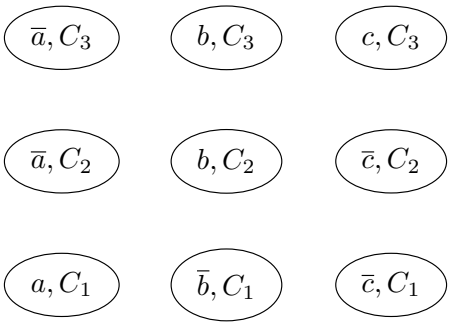


Figura 8.4: *Vértices del nuevo grafo para la reducción.*

- Se añaden aristas al grafo por dos motivos diferentes. Tanto cuando las cláusulas C_i y C_j de los vértices sean la misma (aristas verticales en la Figura 8.5, como cuando sean vértices con literales directa y negada de una misma variable (aristas horizontales).

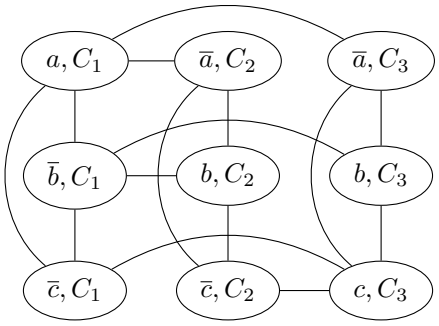


Figura 8.5: *Reducción a* CONJUNTO INDEPENDIENTE *del* 3SAT *de la fórmula* (8.2).

Para comprobar que efectivamente es una reducción hay que ver que el máximo conjunto independiente de nodos del grafo de la Figura 8.5 se corresponde con una combinación de literales que activan la función f de la expresión (8.2). Resolvemos el problema a ojo. Una posible solución, formada por 3 vértices es la mostrada en la Figura 8.6.

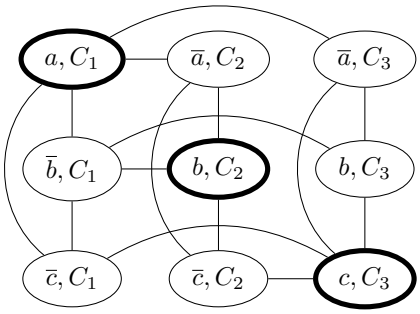


Figura 8.6: *Solución del* CONJUNTO INDEPENDIENTE *que induce una solución para* 3SAT.

Con la solución de la Figura 8.6 tenemos que una solución para el 3SAT de la fórmula (8.2) viene dada por $a = b = c =$ *cierto*. También se hubiera podido tomar la solución mostrada en la Figura 8.7

Con esta otra solución obtenemos que los valores $a = b =$ *cierto* son solución del problema de la fórmula, independientemente del valor asignado a c.

De todo ello trasciende que, si pudiéramos resolver en tiempo polinómico el problema de encontrar el máximo CONJUNTO INDEPENDIENTE de un grafo,

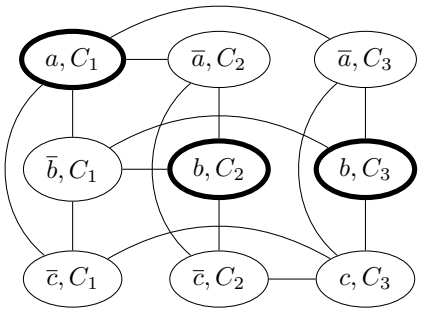


Figura 8.7: *Solución alternativa del* CONJUNTO INDEPENDIENTE *que induce una solución para* 3SAT.

podríamos resolver también polinómicamente 3SAT, y por tanto, SAT.

8.7.2 Reducción de CONJUNTO INDEPENDIENTE a RECUBRIMIENTO POR NODOS

Se presenta a continuación una reducción muy sencilla. El problema del recubrimiento por nodos consiste en encontrar un subconjunto mínimo de nodos de un grafo de manera que tota arista tenga al menos uno de sus extremos dentro del conjunto. La versión decisional es la que se define de la siguiente manera.

Definición 8.10 RECUBRIMIENTO POR NODOS. *Dado un grafo G(V,E) y un número natural k, determinar si existe un conjunto de k nodos en G, tal que toda arista de G tenga al menos un nodo dentro del conjunto.*

La reducción desde CONJUNTO INDEPENDIENTE es inmediata. Llamando $G(V, E)$ a una instancia del CONJUNTO INDEPENDIENTE y $G'(V', E')$ a una del RECUBRIMIENTO POR NODOS, el algoritmo polinómico que constituye la reducción es el siguiente.

- $V' = V$.
- $E' = E$.
- $k = |V| - k$.

En la Figura 8.8 se ilustra la inmediatez de esta reducción. Los nodos en negrita forman el conjunto solución. Se puede verificar en tiempo polinómico que todas las aristas del grafo tienen algún extremo en alguno de esos nodos. Esta solución está obtenida a partir de la solución de la Figura 8.6 utilizando el algoritmo descrito aquí.

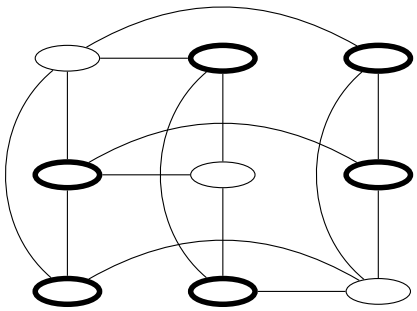


Figura 8.8: *Reducción de* CONJUNTO INDEPENDIENTE *a* RECUBRIMIENTO POR NODOS.

Total, que si supiéramos resolver eficientmente el problema del mínimo recubrimiento por nodos, entonces también resolveríamos eficientmente todos los anteriores.

8.7.3 Reducción de 3SAT a PROGRAMACIÓN LINEAL ENTERA

En términos genéricos se puede enunciar el problema de la programación linial tal como se define aquí.

Definición 8.11 PROGRAMACIÓN LINEAL ENTERA. *Dada una matriz A de $m \times n$ números reales, $A \in \mathbb{R}^{m \times n}$, y un vector $b \in \mathbb{R}^m$, determinar si existe un vector $x \in \mathbb{Z}^n$ tal que $Ax \geq b$.*

De nuevo, para plantear la reducción hay que partir de una instancia del problema 3SAT. Se parte de la misma expresión que en la Sección 8.7.1, habiendo renombrado $x_1 = a$, $x_2 = b$, y $x_3 = c$ por conveniencia en la descripción de la reducción.

$$f = (x_1 \vee \overline{x}_2 \vee \overline{x}_3) \wedge (\overline{x}_1 \vee x_2 \vee \overline{x}_3) \wedge (\overline{x}_1 \vee x_2 \vee x_3) \tag{8.3}$$

Y ahora hay que establecer algún método que a partir de esta instancia sea capaz de plantear una matriz A y un vector b, de manera que la solución del producto $Ax \geq b$ se cumpla para los mismos valores de las variables de la fórmula (8.3).

Si llamamos C_1, C_2 y C_3 a las cláusulas de la fórmula, La reducción es la siguiente.

- $A[i, j] = 1.0$ si x_j aparece en C_i sin negación.

- $A[i,j] = -1.0$ si x_j aparece en C_i con negación.
- $A[i,j] = 0.0$ si x_j no aparece en C_i.
- $b[i] = 1.0 - n_i$ siendo n_i el número de literales negados a C_i.

Rellenando una matriz tal como se prescribe, obtenemos

$$\begin{pmatrix} 1.0 & -1.0 & -1.0 \\ -1.0 & 1.0 & -1.0 \\ -1.0 & 1.0 & 1.0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} -1.0 \\ -1.0 \\ 0.0 \end{pmatrix}$$

Resolviendo esta ecuación nos queda

$$x = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}.$$

Y efectivamente esta solución también satisface la fórmula (8.3).

En este último capítulo se ha hecho una pequeña incursión en el mundo de la Complejidad Algorítmica. Los temas que se han tocado son notablemente distantes a los de cualquier capítulo anterior. Este último trata contenidos más propios del mundo de la investigación. Los anteriores trataban problemas que, todos ellos, se utilizan en el mundo empresarial productivo.

En definitiva se ha visto que para los problemas computacionales que no sabemos resolver en tiempos polinómicos, podemos al menos pseudo ordenarlos según su complejidad. Hemos enunciado algunos problemas difíciles, pero también se ha visto que todos ellos tienen una complejidad parecida, ya que se han mostrado reducciones polinómicas desde el primer problema $\mathcal{NP}$, el 3SAT al problema del CONJUNTO INDEPENDIENTE, y desde éste a RECUBRIMIENTO POR NODOS. Finalmente, también se ha reducido de 3SAT al problema de PROGRAMACIÓN LINEAL ENTERA.

Muchas gracias por vuestra atención.

Carles Franquesa

Barcelona, agosto de 2010.

Apéndice A

Estilo

A lo largo de todo el libro aparecen una gran cantidad de códigos fuente bajo el título genérico de Algoritmo. Todos ellos se entregan en forma de códigos fuente bajo demanda por correo electrónico al autor.

Es bien sabido que los márgenes de liberdad que tolera cualquier lenguaje de programación informática son lo bastante amplios para que haya programadores para todos los gustos. Hay algunas pautas que se siguen en los algoritmos a lo largo de todo el libro. Así pues, para aclarar cuáles son las normas utilizadas en el código proporcionado, seguidamente se dan las directrices.

- En general, todo el codi és en minúsculas. Un criterio de estilo ampliamente difundido es empezar los nombres de las clases en mayúscula, *classe Viajante*. Eso en lenguaje java, es casi obligatorio. De todas maneras, no se considera necesario de cara a la legibilidad ni ninguna otra razón, tener que alternar mayúsculas en medio del código. Se considera que un código no tiene que ser tan complicado de leer como para que sea preciso fijar aspectos abiertos del lenguaje. Al autor, los cambios entre mayúscules y minúsculas le molestan, *classe viajante*.

- Para identificadors de mas de una palabra, se utiliza guiones bajos de separación, *cola_de_prioridad*. También hay quien utiliza una notación que últimamente ha dado en llamarse notación *camell*, que pone en mayúscula cada nueva palabra del mismo identificador *ColaDePrioridad*. Y cosas peores como la notación camello excepto la primera palabra, *colaDePrioritat*. Así se hace en el lenguaje java para las variables y las funciones miembro.

- Se utiliza una sola letra mayúscula para identificar colecciones. Vectores, matrices, colas, etcètera.

- Se utiliza abastamente variables miembro públiques. Arguyendo favoreces

la modularidad, en mucho código que se está construyendo hot día, se utilizan sistemáticamente envoltorios para todas las variables miembro, que se declaran privadas. También son imposiciones provenientes del lenguaje java, que tal vez tengan alguna consecuencia positiva. El autor, no es amigo de los *setters* y los *getters* cuando no tienen ninguna justificación. En concreto, las variables miembro que almacenan resultados de cálculo son declaradas públicas sistemáticamente.

- No se crean clases para estructuras pequeñas. Especialmente las que contenen tan solo miembros de tamaño fijo, como son los tipos primitivos.
- Las funciones miembro muy cortas, se implementan enteras en la misma línea, o en la siguiente, que su cabecera. En este caso, excepcionalmente, las llaves de apertura y cierre están en la misma línea.
- Siempre que se pueda evitar un *else* añadiendo un *return* en el cuerpo del *if* correspondiente se hace. Así se ahorra un nivel de indentación en la parte correspondiente a la que estaría el cuerpo del *else*.
- Los parámetros formales de los constructores cuando sirven para inicializar una variable miembro, siempre tienen exactamente el mismo nombre que las variables que inicializan, con un guión bajo de prefijo. O sea, en los constructores no se usa el puntero *this*.

Todos los algoritmos del libro ocupan una sola página como máximo. Por este motivo, hay normas que se siguen siempre que se puede, y cuando el espacio lo requiere, se infringen.

- Las llaves de apertura y cierre no se ponen cuando el cuerpo que limitan es una instrucció que ocupa una sola línea.
- Las llaves de apertura, en los arxivos de cabecera *.h, se ponen en la misma línea de la cabecera que abren. Para el caso de les funciones en los archivos *.cpp, la llave de apertura ocupa una línea ella sola.
- Las llaves de cierre se ponen siempre que se pueda en líneas diferentes, correctamente indentadas.
- Cada variable se declara en una línea independiente.

Bibliografía

[1] William F. Ames. Numerical method for partial differential equations, section 1.6. *Academic Press, New York*, 1977.

[2] D. L. Applegate, R. E. Bixby, V. Chvátal, and W. J. Cook. *The Traveling Salesman Problem (A Computational Study).* Princeton University Press, 2006.

[3] T. Back. *Evolutionary Algorithms in Theory and Practice.* Oxford, 1996.

[4] Karl-Heinz Borgwardt. *The Simplex Algorithm: A Probabilistic Analysis, Algorithms and Combinatorics.* Springer-Verlag, 1987.

[5] R. Bosch. "Mona Lisa TSP Challenge". *http://www.tsp.gatech.edu/data/ml/monalisa.html*, 2009.

[6] G. Brassard and P. Bratley. *Fundamentos de Algoritmia.* Prentice Hall, 1997.

[7] N. Christofides. Worst-case analysis of a new heuristic for the traveling salesman problem. *Report 388. GSIA. Carnegie-Mellon University*, 1976.

[8] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press, 2001.

[9] P. Crescenzi and V. Kahn. A Compendium of NP Optimization Problems. *http://www.nada.kth.se/ viggo/problemlist/*, 2009.

[10] G. Dantzig, R. Fulkerson, and S. Johnson. Solution of a large-scale traveling-salesman problem. *OPERATIONS RESEARCH*, 2(4):393–410, 1954.

[11] George B. Dantzig and Mukund N. *Linear programming 1: Introduction.* Springer-Verlag., 1997.

[12] Real Academia Española. *Diccionario de la Lengua Española*, volume Vigésimo Primera Edición. ISBN-84-7739-615-9. Dipòsit Legal B.18446-1994. Tercera Edificó, Segona Reimpressió. Juny 1994.

[13] S. Fidanova. Ant Colony Optimization for Multiple Knapsack Problem and Model Bias. *Lecture Notes in Computer Sciences.*, 3401:282–289, 2004.

[14] M. R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman., 1979.

[15] M. Grötschel and M.W. Padberg. *Polyhedral Theory. The Traveling Salesman Problem: A guided Tour of Combinatorial Optimization.* Wiley, Chichester, 1985.

[16] Francis B. Hildebrand. Finite-difference equations and simulations, section 2.2. *Prentice-Halll, Englewood Cliffs, New Jersey*, 1968.

[17] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems.* Springer Verlag, 2005.

[18] E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, and D.B. Shmoys. *The Traveling Salesman Problem.* Wiley-Interscience, 1985.

[19] Evar D. Nering and Albert W. Tucker. *Linear Programs and Related Problems.* Academic Press (elementary, 1993.

[20] B. Novak and S. Schwarz. Vojtech Jarnik (22.12.1897 - 22.9.1970), Acta Arithmetica 20, 1972.

[21] J.J. Salazar. *Programación Matemàtica.* Díaz de Santos, 2001.

[22] R. Sedgewick. *Algorithms in C++. Part 5. Graph Algorithms.* Addison-Wesley, 1992.

[23] R. Sedgewick. *Bundle of Algorithms in C++, Parts 1-5: Fundamentals, Data Structures, Sorting, Searching, and Graph Algorithms (3rd Edition).* Addison-Wesley, 2001.

[24] W. Shakespeare. *Hamlet.* 1603.

[25] M.A. Weiss. *Data Structures and Algorithm Analysis in C++ (2nd Edition).* Addison-Wesley, 1999.

[26] J. Widrow, D.E. Rumelhart, and M.A. Lehr. *Neural networks: Applications in industry, business and science*, volume 37. ACM communications, 1994.

[27] L.A. Wolsey. *Integer Programming.* John Wiley & Sons, 1998.

Algoritmos

Esquemas Algorítmicos

Figuras