# Parte III

# Aplicaciones

CAPÍTULO

# 10 Juegos y diversión

Este capítulo introduce tres técnicas algorítmicas importantes, mostrando su uso en la implementación de dos programas que resuelven problemas recreativos. El primer problema es una sopa de letras que involucra la búsqueda de palabras en un vector bidimensional de caracteres. El segundo es el problema de llevar a cabo una jugada óptima en el juego de las tres en raya.

En este capítulo veremos:

- Cómo usar el algoritmo de búsqueda binaria, modificando el de la Figura 5.12, para incorporar información procedente de búsquedas sin éxito y resolver casos particulares grandes de un problema de búsqueda de palabras en menos de 1 segundo.
- Cómo usar el algoritmo de la *poda alfa-beta* para acelerar el algoritmo recursivo de la Sección 7.7.
- Cómo usar las tablas hash para incrementar la velocidad del algoritmo de las tres en raya.

## 10.1 Sopas de letras

Una *sopa de letras* consiste en buscar palabras en un vector bidimensional de letras. Las palabras pueden estar orientadas en cualquiera de las ocho direcciones.

La entrada para el problema de la *sopa de letras* es un vector bidimensional de caracteres y una lista de palabras. El objetivo es encontrar las palabras en la matriz. Estas palabras pueden aparecer en horizontal, vertical o diagonal en cualquiera de los dos sentidos (hay un total de ocho direcciones). Como ejemplo, la sopa mostrada en la Figura 10.1 contiene las palabras `lado`, `los`, `pez` y `cual`. La palabra `lado` comienza en la fila 0, columna 0, es decir, en el punto (0, 0), y se extiende hasta el (0, 3); la palabra `los` va desde (0, 0) hasta (2, 0); `pez` desde (3, 0) hasta (1, 2); y `cual` desde (3, 3) hasta (0, 0).

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | l | a | d | o |
| 1 | o | a | z | a |
| 2 | s | e | u | m |
| 3 | p | t | r | c |

**Figura 10.1** Ejemplo de sopa de letras.

### 10.1.1 Teoría

El algoritmo que usa la fuerza bruta busca cada palabra de la lista.

Hay varios algoritmos ingenuos que se pueden usar para resolver el problema de la sopa de letras. El más directo es el que hace uso de la fuerza bruta:

```
para cada palabra P en la lista de palabras
    para cada fila F
        para cada columna C
            para cada dirección D
                comprobar si P aparece en la fila F, columna C
                en la dirección D
```

Un algoritmo alternativo busca desde cada punto en la sopa, en cada dirección y para cada longitud de palabra y mira si esa palabra pertenece a la lista.

Puesto que hay 8 direcciones, este algoritmo requiere $8PFC$ comprobaciones. Para las sopas habituales de las revistas, donde encontramos 40 palabras o más en una matriz de 16 filas por 16 columnas, esto supone unas 80.000 comprobaciones, lo cual no es un gran problema para una máquina moderna. Pero si consideramos una variación en la que solamente se proporciona la sopa y la lista de palabras posibles viene dada por aquellas que aparecen en el diccionario, entonces el número de palabras se elevaría hasta 40.000 en lugar de 40, generándose unos 80 millones de comprobaciones. Si se dobla el tamaño de la matriz tendríamos 320 millones de comprobaciones, lo que deja de ser un cálculo trivial. Queremos un algoritmo que pueda resolver una sopa de letras de este tamaño en aproximadamente un segundo. Para ello consideremos el siguiente algoritmo alternativo:

```
para cada fila F
    para cada columna C
        para cada dirección D
            para cada longitud de palabra L
                comprobar si los L caracteres a partir de la
                            posición (F, C) en la dirección
                            D forman una palabra
```

Las comprobaciones se pueden hacer utilizando búsqueda binaria.

Este algoritmo modifica el bucle para evitar la búsqueda de cada palabra de la lista. Supongamos que la longitud de las palabras está limitada a 20 caracteres. En este caso, el número de comprobaciones realizadas por el algoritmo es $160FC$. Para una sopa de 32 filas por 32 columnas, esto supone unas 160.000 comprobaciones. El problema está ahora en que tenemos que decidir si una palabra pertenece a la lista de palabras. Si usamos una búsqueda lineal perdemos todo lo ganado. Pero si usamos una buena estructura de datos, podemos esperar que la búsqueda sea eficiente. Si la lista de palabras está ordenada, lo que es de esperar en un diccionario, entonces podemos usar una búsqueda binaria (mostrada en la Figura 5.12) con lo que para cada comprobación se realizan $\log P$ comparaciones entre cadenas. Para 40.000 palabras, esto significa realizar unas 16 comparaciones por comprobación, dando un total de menos de 3 millones de comparaciones. Esto se puede hacer con toda seguridad en unos pocos segundos. Este algoritmo es mejor que el anterior en un factor de 100.

Este algoritmo se puede mejorar aún más. Supongamos que estamos buscando en alguna dirección y nos damos cuenta de que hemos leído la secuencia de caracteres qx. El diccionario no contiene ninguna palabra comenzando por qx. ¿Vale entonces la pena seguir ejecutando el bucle interno? La respuesta es obviamente no. Si detectamos una secuencia que no es prefijo de ninguna palabra del diccionario, podemos cambiar inmediatamente de dirección. Este algoritmo se expresa de la siguiente forma utilizando pseudocódigo:

Si una secuencia de caracteres no es un prefijo de ninguna palabra del diccionario, podemos terminar la búsqueda en esa dirección.

```
para cada fila F
    para cada columna C
        para cada dirección D
            para cada longitud de palabra L
                comprobar si los L caracteres a partir de la
                             posición (F, C) en la dirección
                             D forman una palabra
                si no forman un prefijo,
                    salir;     // del bucle más interno
```

El único detalle algorítmico pendiente de tratar es la implementación de la comprobación de prefijo. Es decir, suponiendo que la secuencia de caracteres actual no pertenece a la lista de palabras, ¿cómo podemos decidir si es un prefijo de alguna palabra de la lista? La respuesta es bastante simple. El algoritmo de búsqueda binaria de la Figura 5.12 reduce el dominio de búsqueda a un objeto y entonces mira si coincide con el elemento buscado. Supongamos que en lugar de hacer la comprobación, simplemente devolvemos la posición y dejamos que el usuario use la información. Entonces, por supuesto, es fácil para el que hace la llamada comprobar si el elemento en la posición devuelta coincide o no con el

La comprobación de prefijo también se puede hacer usando búsqueda binaria.

```
1  /**
2   * Lleva a cabo la búsqueda binaria para el problema de la
3   * sopa de letras usando una comparación por nivel.
4   * Devuelve la última posición examinada, la cual o bien
5   * contiene a x, o bien x es un prefijo suyo, o bien
6   * no hay ninguna palabra de la que x es prefijo.
7   */
8  private static int busquedaPrefijo( String [ ] a,
9                                      String x, int n )
10 {
11     int inf = 0;
12     int sup = n - 1;
13
14     while( inf < sup )
15     {
16         int med = ( inf + sup ) / 2;
17         if( a[ med ].compareTo( x ) < 0 )
18             inf = med + 1;
19         else
20             sup = med;
21     }
22
23     return inf;
24 }
```

**Figura 10.2** Búsqueda binaria modificada para devolver el último punto de la búsqueda.

elemento buscado. Si no es así, también es fácil ver si la secuencia de caracteres buscada es un prefijo de alguna palabra de la lista, ya que en tal caso debe ser un prefijo de la cadena situada en la posición devuelta (en el Ejercicio 10.3 se le pide que demuestre esto). En la Figura 10.2 mostramos este algoritmo modificado llamado `busquedaPrefijo`.

### 10.1.2 Implementación en Java

Nuestra implementación sigue la descripción del algoritmo.

Nuestra implementación sigue la descripción del algoritmo casi al pie de la letra. Diseñamos una clase llamada `BusquedaPalabras` para almacenar la sopa y la lista de palabras así como los correspondientes canales de entrada. El esqueleto de la clase se muestra en la Figura 10.3. Por simplicidad, asumimos un límite de

```
1  // Clase para resolver el problema de la sopa de letras
2  //
3  // CONSTRUCTOR: sin inicialización
4  // ********************OPERACIONES PÚBLICAS*********************
5  // int resolverSopa( ) --> Imprime todas las palabras encontradas
6  //                         en la sopa; devuelve el número de palabras
7
8  public class BusquedaPalabras
9  {
10     private static final int MAX_FILAS    =      64;
11     private static final int MAX_COLUMNAS =      64;
12     private static final int MAX_PALABRAS = 100000;
13
14     public BusquedaPalabras( )
15       { /* Figura 10.4 */ }
16     public int resolverSopa( )
17       { /* Figura 10.8 */ }
18
19     private int filas;
20     private int columnas;
21     private int numPalabras;
22     private String [ ] palabras = new String[ MAX_PALABRAS ];
23     private BufferedReader sopaStream;
24     private BufferedReader palabraStream;
25     private char sopa[ ][ ] =
26                     new char[ MAX_FILAS ][ MAX_COLUMNAS ];
27     private BufferedReader in = new
28          BufferedReader( new InputStreamReader( System.in ) );
29
30     private static int busquedaPrefijo( String [ ] a,
31                                         String x, int n )
32       { /* Figura 10.2 */ }
33     private BufferedReader abreFichero( String mensaje )
34       { /* Figura 10.5 */ }
35     private void leePalabras( )
36       { /* Figura 10.6 */ }
37     private void leeSopa( )
38       { /* Figura 10.7 */ }
39     private int resolverDireccion( int filaBase, int colBase,
40                                    int filaDelta, int colDelta )
41       { /* Figura 10.9 */ }
42 }
```

**Figura 10.3** Esqueleto de la clase `BusquedaPalabras`.

64 filas y columnas para la sopa, y un tamaño máximo de diccionario de 100.000 palabras, dejando el problema de suprimir estas restricciones para el Ejercicio 10.6. La parte pública de la clase consta de un constructor y un único método `resolverSopa`. La sección privada incluye los atributos y las rutinas de soporte.

La Figura 10.4 proporciona el código del constructor, que simplemente abre y lee los dos ficheros correspondientes a la sopa de letras y a la lista de palabras. La rutina de soporte `abreFichero`, mostrada en la Figura 10.5, solicita repetidamente el nombre de un fichero hasta que la apertura tiene éxito. La rutina

El constructor abre los ficheros y lee los atributos. Por brevedad, evitamos algunas comprobaciones de error.

```
1  /**
2   * Constructor para la clase BusquedaPalabras. Solicita
3   * el nombre de los ficheros de la sopa y del diccionario.
4   */
5  public BusquedaPalabras( )
6  {
7      sopaStream = abreFichero( "Introduzca fichero de la sopa" );
8      palabraStream = abreFichero( "y el del diccionario " );
9      leeSopa( );
10     leePalabras( );
11 }
```

**Figura 10.4** Constructor de `BusquedaPalabras`.

```
1  /**
2   * Imprime un mensaje y abre un fichero.
3   * Lo intenta repetidas veces hasta que consigue abrirlo.
4   * El programa termina cuando se alcanza el final de fichero.
5   */
6  private BufferedReader abreFichero( String mensaje )
7  {
8      String nombreFichero = "";
9      FileReader fichero;
10     BufferedReader fichEntrada = null;
11
12     do
13     {
14         System.out.println( mensaje + ": " );
15
16         try
17         {
18             nombreFichero = in.readLine( );
19             if( nombreFichero == null )
20                 System.exit( 0 );
21             fichero = new FileReader( nombreFichero );
22             fichEntrada = new BufferedReader( fichero );
23         }
24         catch( IOException e )
25           { System.err.println( "No se puede abrir " + nombreFichero ); }
26     } while( fichEntrada == null );
27
28     System.out.println( "Abierto " + nombreFichero );
29     return fichEntrada;
30 }
```

**Figura 10.5** Rutina `abreFichero` para abrir los ficheros de la sopa de letras y de la lista de palabras.

`leePalabras`, mostrada en la Figura 10.6, lee la lista de palabras. La mayor parte del código está relacionada con la comprobación de errores: no queremos leer más de `MAX_PALABRAS` palabras, y queremos asegurarnos de que la lista de palabras está ordenada. Análogamente, la rutina `leeSopa`, mostrada en la Figura 10.7, lee la sopa de letras y también se encarga de la comprobación de errores. Necesitamos asegurarnos de que podemos manejar sopas vacías, y queremos avisar al usuario si la sopa no es rectangular. En interés de la mayor brevedad posible, se han omitido algunas de las comprobaciones que deberían haberse realizado. En el Ejercicio 10.1 se le pide que averigue lo que falta.

Usamos dos bucles para recorrer las ocho direcciones.

La rutina `resolverSopa` de la Figura 10.8 anida los bucles que recorren las filas, las columnas y las direcciones, y llama a la rutina privada `resolverDireccion` para cada una de las posibilidades. El valor devuelto es el número de palabras encontrado. Una dirección viene dada por una dirección en la columna y una dirección en la fila. Por ejemplo, el sur viene indicado por `dc = 0` y `df = 1`, y el noreste por `dc = 1` y `df = -1`. Las variables `dc` y `df` toman valores entre `-1` y `1`, pero no pueden tomar simultáneamente el valor `0`. Ahora, lo que queda por escribir es `resolverDireccion`, que aparece en la Figura 10.9.

La rutina `resolverDireccion` construye una cadena partiendo de la fila y la columna base y avanzando en la dirección apropiada.

```
1  /**
2   * Rutina para leer el diccionario.
3   * Se muestra un mensaje de error si no está ordenado.
4   * Se hace una comprobación para no sobrepasar MAX_PALABRAS.
5   */
6  private void leePalabras( )
7  {
8      numPalabras = 0;
9
10     try
11     {
12         while( ( palabras[ numPalabras ] =
13                         palabraStream.readLine( ) ) != null )
14         {
15             if( numPalabras != 0 && palabras[ numPalabras ].
16               compareTo( palabras[ numPalabras - 1 ] ) < 0 )
17             {
18                 System.err.println( "El diccionario no está " +
19                                     "ordenado - saliendo" );
20                 continue;
21             }
22             else if( ++numPalabras >= MAX_PALABRAS )
23                 break;
24         }
25
26         if( palabraStream.ready( ) )
27             System.err.println( "Aviso: no se han leido los " +
28                                 "datos - incremente MAX_PALABRAS" );
29     }
30     catch( IOException e ) { }
31 }
```

**Figura 10.6** Rutina `leePalabras` para leer la lista de palabras.

```
1  /**
2   * Rutina para leer la sopa de letras.
3   * Comprueba que la sopa es rectangular.
4   * No se comprueba si se excede la capacidad.
5   */
6  private void leeSopa( )
7  {
8      String unaLinea;
9
10     try
11     {
12         unaLinea = sopaStream.readLine( );
13         if( unaLinea == null )
14         {
15             filas = 0;
16             return;
17         }
18         columnas = unaLinea.length( );
19         for( int i = 0; i < columnas; i++ )
20            sopa[ 0 ][ i ] = unaLinea.charAt( i );
21
22         for( filas = 1;
23              ( unaLinea = sopaStream.readLine( ) ) != null;
24                        filas++ )
25         {
26             if( unaLinea.length( ) != columnas )
27                 System.err.println( "La sopa es incorrecta" );
28
29             for( int i = 0; i < columnas; i++ )
30                sopa[ filas ][ i ] = unaLinea.charAt( i );
31         }
32     }
33     catch( IOException e ) { }
34 }
```

**Figura 10.7** Rutina `leeSopa` para leer la sopa de letras.

```
1  /**
2   * Rutina para resolver una sopa de letras.
3   * Lleva a cabo comprobaciones en todas la direcciones.
4   * @return número de palabras encontradas
5   */
6  public int resolverSopa( )
7  {
8      int numPal = 0;
9
10     for( int f = 0; f < filas; f++ )
11         for( int c = 0; c < columnas; c++ )
12             for( int df = -1; df <= 1; df++ )
13                 for( int dc = -1; dc <= 1; dc++ )
14                     if( df != 0 || dc != 0 )
15                         numPal += resolverDireccion( f,
16                                                      c, df, dc );
17
18     return numPal;
19 }
```

**Figura 10.8** Rutina `resolverSopa` que busca en todas las direcciones a partir de un punto.

```
1  /**
2   * Busca en la sopa a partir de un punto y en una dirección.
3   * @return número de palabras encontradas
4   */
5  private int resolverDireccion( int filaBase, int colBase,
6                                 int filaDelta, int colDelta )
7  {
8      String secuencia = "";
9      int numPal = 0;
10     int resultadoBusqueda;
11
12     secuencia += sopa[ filaBase ][ colBase ];
13
14     for( int i = filaBase + filaDelta, j = colBase + colDelta;
15              i >= 0 && j >= 0 && i < filas && j < columnas;
16              i += filaDelta, j += colDelta )
17     {
18         secuencia += sopa[ i ][ j ];
19         resultadoBusqueda = busquedaPrefijo( palabras,
20                                              secuencia, numPal );
21
22         if( !palabras[ resultadoBusqueda ].
23                                 startsWith( secuencia ) )
24           break;
25
26         if( palabras[ resultadoBusqueda ].equals( secuencia ) )
27         {
28             numPal++;
29             System.out.println( "Encontrada " + secuencia +
30                     " de " + filaBase + " " + colBase +
31                     " a " + i + " " + j );
32         }
33     }
34     return numPal;
35 }
```

**Figura 10.9** Implementación de una búsqueda individual.

Asumimos que las palabras de una letra no están permitidas (pues de estarlo se contarían ocho veces). En las líneas 15 y 16, extendemos la cadena mientras comprobamos que no rebasamos los límites de la sopa. En la línea 18 añadimos el siguiente carácter y en las líneas 19 y 20 llevamos a cabo una búsqueda binaria. Si la cadena leída hasta el momento no es un prefijo de ninguna palabra, podemos parar de mirar en esa dirección. En caso contrario, sabemos que continuaremos haciéndolo, tras comprobar, en la línea 26, si ya se ha leído una palabra de la lista. La línea 34 devuelve el número de palabras encontradas finalmente. En la Figura 10.10 se muestra un programa `main` simple.

```
1    // main simple
2  public static void main( String [ ] args )
3  {
4      BusquedaPalabras p = new BusquedaPalabras( );
5      System.out.println( "Resolviendo..." );
6      p.resolverSopa( );
7  }
```

**Figura 10.10** Rutina `main` simple para el problema de la sopa de letras.

## 10.2 El juego de las tres en raya

Recordemos el algoritmo simple de la Sección 7.7 que permitía al computador elegir el movimiento óptimo en el juego de las tres en raya. La estrategia conocida como *minimax* consistía en lo siguiente:

1. Si el estado es *terminal* (es decir, se puede evaluar inmediatamente), devolvemos su valor.
2. En caso contrario, si le toca mover al computador, devolvemos el valor máximo de todos los estados que se pueden alcanzar después de hacer un movimiento. Los valores alcanzados se calculan de forma recursiva.
3. En caso contrario, es el turno de la persona. Devolvemos entonces el mínimo de todos los estados alcanzables después de hacer un movimiento. De nuevo, los valores alcanzados se calculan de forma recursiva.

La *estrategia minimax* examina una gran cantidad de estados. Podemos arreglárnoslas con menos, sin perder por ello ninguna información.

### 10.2.1 Poda alfa-beta

Aunque la estrategia minimax proporciona un movimiento óptimo para las tres en raya, lleva a cabo una gran cantidad de búsquedas. En concreto, para decidir el primer movimiento hace alrededor de medio millón de llamadas recursivas. Esto sucede porque hace más búsquedas de las imprescindibles. Supongamos que el computador está considerando cinco movimientos: $C_1$, $C_2$, $C_3$, $C_4$ y $C_5$. Supongamos que la evaluación recursiva de $C_1$ revela que $C_1$ fuerza un empate. A continuación se evalúa $C_2$. En este momento, llegamos a un estado en el que le toca mover al humano. Supongamos que en respuesta a $C_2$, el humano puede considerar $H_{2A}$, $H_{2B}$, $H_{2C}$ y $H_{2D}$, y además que una evaluación de $H_{2A}$ muestra que se ha forzado un empate. Automáticamente se ha visto que $C_2$ es un empate en el mejor de los casos e incluso posiblemente una derrota para el computador (porque se supone que el humano juega óptimamente). Puesto que necesitamos mejorar $C_1$, no tenemos que evaluar $H_{2B}$, $H_{2C}$ ni $H_{2D}$. Decimos que $H_{2A}$ es una *refutación*, en el sentido de que $C_2$ no es un movimiento mejor que el que ya hemos visto. En consecuencia devolvemos que $C_2$ es un empate y mantenemos a $C_1$ como el mejor movimiento hasta el momento, lo cual se muestra en la Figura 10.11.

Una *refutación* es un posible movimiento del contrario que demuestra que el movimiento original que estamos examinando no es una mejora con respecto a otros previamente considerados. Si encontramos una refutación, no tenemos que examinar más movimientos en las alternativas por debajo de la original y se pueden dar por concluidas las llamadas recursivas en ejecución.

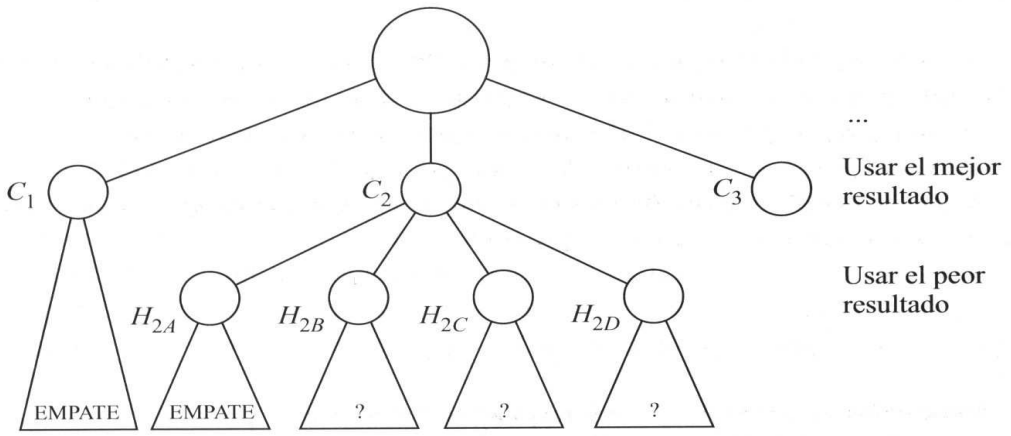



**Figura 10.11** Poda alfa-beta: después de que se haya evaluado $H_{2A}$, $C_2$, que es el menor de los $H_2$, es un empate en el mejor de los casos, por lo que no puede ser una mejora respecto a $C_1$. Por tanto, no necesitamos evaluar $H_{2B}$, $H_{2C}$ y $H_{2D}$, y se puede pasar directamente a $C_3$.

La *poda alfa-beta* se usa para reducir el número de estados evaluados en una búsqueda minimax. Alfa es el valor que el humano tiene que rechazar y beta el valor que el computador tiene que rechazar.

No necesitamos evaluar cada nodo completamente; para algunos nodos basta con encontrar una refutación. Esto significa que algunos bucles pueden terminar antes de tiempo. En concreto, cuando el humano evalúa una posición, como $C_2$, encontrar una refutación es tan bueno como el mejor movimiento absoluto. La misma lógica se aplica al computador. En cualquier punto, `alfa` es el valor que el humano ha de rechazar, y `beta` es el valor que el computador ha de rechazar. Cuando el humano hace una búsqueda, cualquier movimiento menor que `alfa` es equivalente a `alfa`; si la búsqueda la realiza el computador entonces cualquier movimiento mayor que `beta` es equivalente a `beta`. Esta estrategia recibe el nombre de *poda alfa-beta*.

Como se muestra en la Figura 10.12, la poda alfa-beta se obtiene realizando solamente unos pocos cambios en `elegirMovimiento`. Tanto `alfa` como `beta` se pasan como parámetros adicionales. Inicialmente, se llama a `elegirMovimiento` con valores `HUMANO_GANA` y `COMPUTADORA_GANA` para `alfa` y `beta` respectivamente. Las líneas 16 a 20 reflejan un cambio en la inicialización de `valor`. La evaluación de un movimiento es sólo ligeramente más compleja que la original de la Figura 7.27. La llamada recursiva de la línea 29 incluye los parámetros `alfa` y `beta`, cuyos valores se ajustan en las línea 36 o 38, cuando ello es necesario. El otro cambio aparece en la línea 42, que provoca la terminación por anticipado cuando se encuentra una refutación.

La poda alfa-beta produce más mejoras cuando encuentra pronto refutaciones.

Para beneficiarse lo máximo posible de las ventajas de la poda alfa-beta, los programas de juegos normalmente intentan aplicar heurísticas para colocar los mejores movimientos al comienzo de la búsqueda. Esto produce más poda que la esperada en una búsqueda aleatoria de estados. En la práctica, la poda alfa-beta limita la búsqueda a solamente $O(\sqrt{N})$ nodos, donde $N$ es el número de nodos que se examinarían sin poda alfa-beta. Esto supone una gran cantidad de ahorro. El ejemplo de las tres en raya no es ideal al respecto porque los valores son demasiado similares entre sí, pero aun así la búsqueda inicial se reduce a unos 18.000 estados.

### 10.2.2 Tablas de transposición

Una *tabla de transposición* almacena estados previamente evaluados.

Otra práctica empleada normalmente consiste en usar una tabla para guardar todos los estados que ya han sido evaluados. Por ejemplo, en el curso de la búsqueda del primer movimiento, el programa examinará los estados mostrados en la Figura 10.13. Si se guardan los valores de los estados, la segunda vez que aparezca uno de ellos no será necesario calcular su valor de nuevo, lo que hace que dicho estado se convierta en terminal. La estructura que almacena los estados recibe el nombre de *tabla de transposición*, la cual se implementa usando una tabla hash. En muchos casos, la aplicación de esta técnica, análoga a la implementación de la programación dinámica que vimos en la Sección 7.6, puede ahorrar una gran cantidad de cálculos.

Para implementar la tabla de transposición se usa una tabla hash.

Para implementar la tabla de transposición, definimos primero una clase `Estado`, mostrada en la Figura 10.14, que se usa para almacenar cada estado y su valor calculado. Esta clase implementa el interfaz de `Hashable`. También proporcionamos un constructor que se puede inicializar con un tablero (vector bidimensional).

```
1       // Encuentra el movimiento óptimo
2   private Mejor elegirMovimiento( int lado, int alfa, int beta )
3   {
4       int op;                 // El oponente
5       Mejor replica;          // Mejor réplica del oponente
6       int evalSimple;         // Resultado de una evaluación inmediata
7       int mejorFila = 0;
8       int mejorColumna = 0;
9       int valor;
10
11      if( ( evalSimple = valorEstado( ) ) != INCIERTO )
12          return new Mejor( evalSimple );
13
14      if( lado == COMPUTADORA )
15      {
16          op = HUMANO; valor=alfa;
17      }
18      else
19      {
20          op = COMPUTADORA; valor = beta;
21      }
22
23  Exterior:
24      for( int fila = 0; fila < 3; fila++ )
25          for( int columna = 0; columna < 3; columna++ )
26              if( casillaVacia ( fila, columna ) )
27              {
28                  lugar( fila, columna, lado );
29                  replica = elegirMovimiento( op, alfa, beta );
30                  lugar( fila, columna, VACIA );
31                  if( lado == COMPUTADORA && replica.valor > valor
32                      || lado == HUMANO && replica.valor < valor )
33                  {
34                          // Se ha encontrado el mejor movimiento
35                      if( lado == COMPUTADORA )
36                          alfa = valor = replica.valor;
37                      else
38                          beta = valor = replica.valor;
39
40                      mejorFila = fila;
41                      mejorColumna = columna;
42                      if( alfa >= beta )
43                          break Exterior; // Refutación
44                  }
45              }
46
47      return new Mejor( valor, mejorFila, mejorColumna );
48  }
```

**Figura 10.12** Rutina `elegirMovimiento` para calcular el movimiento óptimo usando poda alfa-beta.

La rutina `elegirMovimiento` tiene tres parámetros adicionales: los valores de `alfa` y `beta` y la profundidad actual de la búsqueda.

Los cambios necesarios en la clase `TresEnRaya` son mínimos y se muestran en la Figura 10.15. Se añade un nuevo atributo en la línea 3 y se modifica la declaración de `elegirMovimiento`. Ahora pasamos `alfa` y `beta` (igual que hicimos en la poda alfa-beta) y también la `profundidad` de la recursión, que es inicialmente cero. La llamada inicial a `elegirMovimiento` aparece en las líneas 7 y 8.

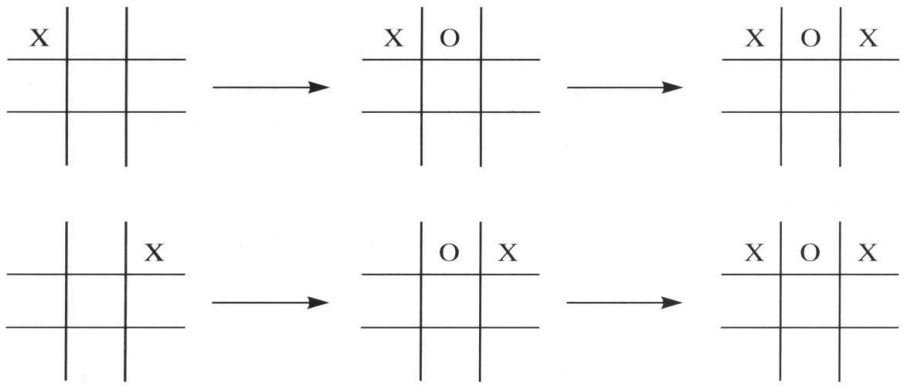



**Figura 10.13** Dos búsquedas que llegan a estados idénticos.

```
1  final class Estado implements Hashable
2  {
3      int [ ][ ] tablero;
4      int valor;
5
6      Estado( int elTablero[ ][ ] )
7      {
8          tablero = new int[ 3 ][ 3 ];
9          for( int i = 0; i < 3; i++ )
10             for( int j = 0; j < 3; j++ )
11                 tablero[ i ][ j ] = elTablero[ i ][ j ];
12     }
13
14     public boolean equals( Object lder )
15     {
16         for( int i = 0; i < 3; i++ )
17             for( int j = 0; j < 3; j++ )
18                 if( tablero[ i ][ j ] !=
19                             ((Estado)lder).tablero[ i ][ j ] )
20                     return false;
21         return true;
22     }
23
24     public int hash( int tamTablero )
25     {
26         int valorHash = 0;
27
28         for( int i = 0; i < 3; i++ )
29             for( int j = 0; j < 3; j++ )
30                 valorHash = valorHash * 4 + tablero[ i ][ j ];
31
32         return valorHash % tamTablero;
33     }
34 }
```

**Figura 10.14** Clase `Estado`.

En la tabla de transposición no almacenamos los estados cercanos a los casos base de la recursión.

Controlamos la profundidad porque no vale la pena incluir todos los estados en la tabla de transposición. La sobrecarga de mantener la tabla sugiere que los estados cercanos a los casos base no deberían guardarse por las siguientes razones:

```
1  class TresEnRaya
2  {
3      private TablaHash transposicion =
4                              new TablaExploracionCuadratica( );
5      publicMejor elegirMovimiento( int lado )
6      {
7          return elegirMovimiento( lado, HUMANO_GANA,
8                                   COMPUTADORA_GANA, 0 );
9      }
10     ...
11 }
```

**Figura 10.15** Cambios en la clase `TresEnRaya` para incorporar la tabla de transposición y la poda alfa-beta.

- Hay demasiados estados.
- La idea de la poda alfa-beta y de la tabla de transposición es reducir los tiempos de búsqueda evitando llamadas lo más pronto posible en el juego: ahorrarnos una llamada muy profunda en la búsqueda no reduce un gran número de estados a examinar ya que en cualquier caso dicha llamada solamente examinará unos pocos estados.

Las Figuras 10.16 y 10.17 muestran el nuevo `elegirMovimiento`. En la línea 8 declaramos un objeto `Estado` llamado `estadoActual`. En su momento este objeto se colocará en la tabla de transposición. La variable `profTabla` nos dirá la profundidad hasta la que permitimos guardar estados en la tabla de transposición. Experimentando, averiguamos que 5 es un valor óptimo. Permitir que se guarden estados examinados en `profundidad` 6 es perjudicial, debido a que el coste extra que supone mantener la tabla de transposición no es compensado por el menor número de estados examinados.

El código tiene algunos trucos pero nada particulamente notable.

Las líneas 17 a 28 son nuevas. Si estamos en la primera llamada a `elegirMovimiento`, inicializamos la tabla de transposición. En caso contrario, si estamos en una profundidad apropiada, comprobamos si ya se ha evaluado el estado, y en tal caso devolvemos su valor. El código tiene dos trucos. En primer lugar, solamente hemos de buscar en la tabla de transposición si `profundidad` es mayor que 3, como se observa en la Figura 10.13. La otra diferencia aparece de la línea 62 en adelante. Inmediatamente antes de devolver el resultado, almacenamos el valor del estado en la tabla de transposición.

El uso de la tabla de transposición en este algoritmo de las tres en raya elimina alrededor de la mitad de los estados a tener en cuenta, con solamente un pequeño coste para las operaciones de la tabla de transposición. Esto significa que prácticamente se dobla la velocidad del programa.

## 10.2.3 El ajedrez

No nos podemos basar en buscar posiciones terminales en el ajedrez. En los mejores programas, se incorpora un conocimiento considerable en la función de evaluación.

En un juego complejo como el ajedrez o el juego Go, no es factible recorrer todo el camino hasta los nodos terminales: algunas estimaciones nos dicen que hay unos $10^{100}$ estados legales en el ajedrez, y ni todos los trucos del mundo reducirían esta cantidad a un nivel manejable. En este caso, tenemos que parar la búsqueda a partir de una cierta profundidad de la recursión. Los nodos en los que se termina la exploración se convierten en nodos terminales, los cuales se evalúan

```
1       // Encuentra el movimiento óptimo
2  private Mejor elegirMovimiento( int lado, int alfa, int beta,
3                                  int profundidad )
4  {
5       int op;             // El oponente
6       Mejor replica;      // Mejor réplica del oponente
7       int evalSimple;     // Resultado de una evaluación inmediata
8       Estado estadoActual = new Estado( tablero );
9       int profTabla = 5;// Máxima profundidad a guardar
10      int mejorFila = 0;
11      int mejorColumna = 0;
12      int valor;
13
14      if( ( evalSimple = valorEstado( ) ) != INCIERTO )
15          return new Mejor( evalSimple );
16
17      if( profundidad == 0 )
18          transposicion.vaciar( );
19      else if( profundidad >= 3 && profundidad <= profTabla )
20      {
21          try
22          {
23              Estado buscaValor = (Estado)
24                      transposicion.buscar( estadoActual );
25              return new Mejor( buscaValor.valor );
26          }
27          catch( ElementoNoEncontrado e ) { } /* Evaluado debajo */
28      }
29
30      if( lado == COMPUTADORA )
31      {
32          op = HUMANO; valor = alfa;
33      }
34      else
35      {
36          op = COMPUTADORA; valor = beta;
37      }
38
39  Exterior:
40      for( int fila = 0; fila < 3; fila++ )
41          for( int columna = 0; columna < 3; columna++ )
42              if( casillaVacia ( fila, columna ) )
43              {
44                  lugar( fila, columna, lado );
45                  replica = elegirMovimiento( op, alfa, beta,
46                                              profundidad + 1 );
47                  lugar( fila, columna, VACIA );
```

**Figura 10.16** Algoritmo de las tres en raya con poda alfa-beta y tabla de transposición (parte 1).

mediante una función que trata de estimar el valor del estado. Por ejemplo, en un programa de ajedrez, la función de evaluación calcula la cantidad relativa y fuerza de las piezas, así como otros factores relacionados con la posición.

Los mejores programas de ajedrez juegan al nivel de gran maestro.

Los computadores son especialmente hábiles en movimientos que implican profundas combinaciones que terminan en un intercambio de material, debido a que es sencillo evaluar la fuerza de las piezas. Sin embargo, extender la profundi-

```
48                  if( lado == COMPUTADORA && replica.valor > valor
49                      || lado == HUMANO && replica.valor < valor )
50                  {
51                      if( lado == COMPUTADORA )
52                          alfa = valor = replica.valor;
53                      else
54                          beta = valor = replica.valor;
55
56                      mejorFila = fila; mejorColumna = columna;
57                      if( alfa >= beta )
58                          break Exterior; // Refutación
59                  }
60              }
61
62          // Inserta en la tabla de transposición
63      estadoActual.valor = valor;
64      if( profundidad <= profTabla )
65          transposicion.insertar( estadoActual );
66
67      return new Mejor( valor, mejorFila, mejorColumna );
68  }
```

**Figura 10.17** Algoritmo de las tres en raya con poda alfa-beta y tabla de transposición (parte 2).

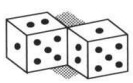
dad de la búsqueda solamente un nivel, supone un incremento de la velocidad de procesamiento de un factor de seis (porque el número de estados se incrementa en un factor de 36). Cada nivel extra de búsqueda refuerza en gran manera la habilidad del programa sólo hasta un cierto límite, que parece haber sido ya alcanzado por los mejores programas. Por otra parte, los computadores no son tan buenas en juegos posicionales más estáticos en los que se requieren evaluaciones más sutiles y un mayor conocimiento del juego. Sin embargo, esto sólo se pone en evidencia cuando el computador está compitiendo con un oponente muy hábil. Los populares programas de ajedrez son hoy en día mejores que la mayoría de los jugadores.

En 1997, el programa *Deep Blue*, fue capaz de derrotar al campeón de ajedrez en un encuentro de seis partidos, usando una enorme capacidad computacional (evaluando unos 200 millones de movimientos por segundo).

## Resumen

Este capítulo ha proporcionado una aplicación de la búsqueda binaria y de algunas técnicas algorítmicas que son usadas habitualmente en programas de juegos como el ajedrez, las damas y Otelo. Los mejores programas para estos juegos tienen el nivel de los mejores expertos en ellos. Sin embargo, el juego Go parece ser demasiado complejo para ser bien jugado a base de búsquedas por parte de un computador.

## Elementos del juego



**estrategia minimax** Estrategia recursiva que permite al computador seleccionar un movimiento óptimo en el juego de las tres en raya.

**poda alfa-beta** Técnica usada para reducir el número de estados evaluados en una búsqueda minimax. Alfa es el valor que el humano ha de rechazar, y beta el valor que el computador ha de rechazar.

**posición terminal** Posición en un juego que se puede evaluar inmediatamente.

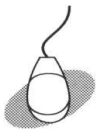
**refutación** Movimiento del contrario que demuestra que el movimiento original en examen no es una mejora de otros considerados previamente. Si encontramos una refutación, no tenemos que examinar más movimientos por debajo del nivel original y las llamadas recursivas en ejecución terminan.

**sopa de letras** Programa que lleva a cabo la búsqueda de palabras en un vector bidimensional de letras. Las palabras pueden estar orientadas en ocho direcciones.

**tabla de transposición** Tabla hash que almacena estados previamente evaluados.

## Errores comunes

1. Cuando use una tabla de transposición, limite el número de posiciones almacenadas para evitar agotar la memoria.
2. Es importante comprobar que se cumplen cada una de nuestras suposiciones. Por ejemplo, en la sopa de letras, compruebe que el diccionario está ordenado. Es un error bastante común olvidar esta comprobación.



## En Internet

Se proporciona el código completo de los dos ejemplos estudiados, aunque el interfaz del juego de las tres en raya deja bastante que desear. Ambos se encuentran en el directorio **Chapter10**. A continuación se enumeran los nombres de los ficheros:

| | |
|---|---|
| **WordSrch.java** | Contiene el algoritmo de la sopa de letras. Se ha traducido aquí por `BusquedaPalabras`. |
| **Best.java** | Contiene la clase `Best`, traducida por `Mejor`, que forma parte del algoritmo de las tres en raya. |
| **TicTacToe.java** | Contiene una clase `TicTacToe`, traducida por `TresEnRaya`. Utiliza la poda alfa-beta y una tabla de transposición. |
| **TicTacMain.java** | Contiene una simple interfaz gráfica de usuario para las tres en raya. |

## Ejercicios

### *Cuestiones breves*

**10.1.** ¿Qué comprobaciones de error faltan en la Figura 10.7?

**10.2.** Para la situación presentada en la Figura 10.18, determine:

*a*) ¿Cuál de las respuestas al movimiento $C_2$ es una refutación?

*b*) ¿Cuál es el valor del estado?

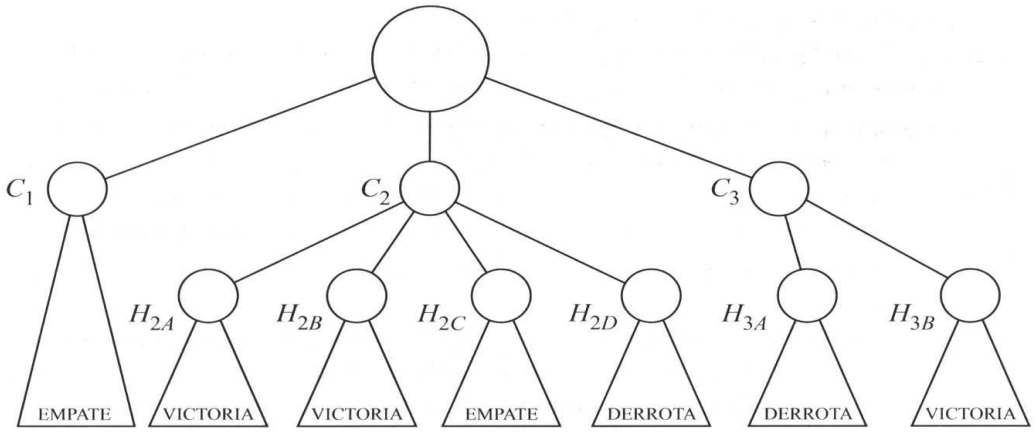



**Figura 10.18** Ejemplo de poda alfa-beta para el Ejercicio 10.2.

## *Problemas teóricos*

**10.3.** Verifiqe que si `x` es un prefijo de alguna palabra de un vector `a`, entonces `x` es un prefijo de la palabra en la que termina la búsqueda binaria.

**10.4.** Explique cómo cambia el tiempo de ejecución del algoritmo de la búsqueda de palabras cuando sucede lo siguiente:

*a*) Se dobla el número de palabras.
*b*) Se doblan simultáneamente el número de filas y columnas.

**10.5.** Describa el efecto de escribir un método `Equals` en lugar de un método `equals`, para la clase `Estado`.

## *Problemas prácticos*

**10.6.** Elimine las restricciones impuestas al tamaño de la sopa de letras y al del diccionario, presentes en la implementación de la clase `BusquedaPalabras`.

**10.7.** Sustituya la búsqueda binaria por una secuencial en el problema de la búsqueda de palabras. ¿Cómo afecta esto a la eficiencia?

**10.8.** Compare la eficiencia del algoritmo de búsqueda de palabras con y sin la búsqueda prefijo.

**10.9.** Implemente una buena interfaz gráfica de usuario para el programa de las tres en raya. Su programa debe trabajar como una aplicación y como un applet.

**10.10.** Un programa para las tres en raya sin interfaz gráfica de usuario necesita código adicional para leer los movimientos y mostrar el tablero. Escriba una interfaz amigable al usuario.

**10.11.** Incluso cuando el computador tiene un movimiento que proporciona una victoria inmediata, puede que no lo realice si detecta que otro movimiento también le llevará a ganar. Algunos de los primeros programas de ajedrez tenían el problema de que entraban en una repetición de estado

cuando detectaban una victoria segura, permitiendo al contrario proclamar un empate. En las tres en raya esto no es un problema porque el programa eventualmente ganará. Modifique el programa de las tres en raya de forma que cuando se encuentre un estado ganador, siempre tome el movimiento que lleve a la victoria más rápidamente. Puede hacer esto sumando `9 - profundidad` a `COMPUTADORA_GANA` de forma que la victoria más rápida produzca el mayor valor.

**10.12.** Use un `int` en lugar de un vector bidimensional para almacenar el tablero en 18 bits. Esto claramente ahorra espacio. ¿Ahorra también tiempo?

**10.13.** Compare la eficiencia del programa de las tres en raya con y sin poda alfa-beta.

**10.14.** Implemente el algoritmo de las tres en raya y mida la eficiencia cuando se permite almacenar valores en la tabla de transposición a distintas profundidades. Mídala también cuando no se usa tabla de transposición. ¿Cómo se ven afectados los resultados por la poda alfa-beta?

### *Prácticas de programación*

**10.15.** Escriba un programa para jugar a las cuatro en raya en un tablero de cinco por cinco. ¿Puede realizar la búsqueda hasta los nodos terminales?

**10.16.** El juego del boggle consiste en una matriz de letras y una lista de palabras. El objetivo es encontrar palabras en la matriz con la restricción de que dos letras seguidas en las palabras deben estar adyacentes en la matriz (es decir, hacia el norte, sur, este u oeste) y de que cada letra de la matriz sólo se puede usar una vez por palabra. Escriba un programa para jugar al boggle.

**10.17.** Escriba un programa para jugar a MAXIT. El tablero es una matriz $N \times N$ de números colocados aleatoriamente al principio del juego. Se designa una posición como posición inicial actual. Dos jugadores se van alternando. En cada turno, un jugador debe seleccionar un elemento de la matriz en la fila o columna actuales. El valor de la posición seleccionada se suma a la puntuación del jugador, y dicha posición se convierte en la actual y no se puede elegir de nuevo. Los jugadores se alternan hasta que se han seleccionado todos los elementos de la fila y columna actuales, en cuyo momento termina el juego ganando el jugador con la mayor puntuación.

**10.18.** Otelo jugado sobre un tablero de 6 por 6 conduce a victoria segura a las fichas negras. Demuestre esto escribiendo un programa. ¿Cuál es la puntuación final si los dos jugadores juegan de forma óptima?

## Bibliografía

Si está interesado en juegos de computador, un buen punto de partida es el siguiente artículo, contenido en un número especial dedicado exclusivamente al tema. Encontrará gran cantidad de información y referencias a otros trabajos sobre ajedrez, damas y otros juegos de computador.

1. K. Lee y S. Mahajan, «The Development of a World Class Othello Program», *Artificial Intelligence* **43** (1990), 21-36.