MANUAL DE LA IMPLEMENTACIÓN

Enero, 1999
Pedro Pablo Gómez Martín
Marco Antonio Gómez Martín
Francisco Javier Cabello Torres

1.- ANATOMÍA DE LOS PROGRAMAS DE JUEGOS

Los juegos son un tipo particular de problemas en los que el modo de solucionarlos no depende únicamente de uno mismo, sino que existe un agente exterior que intenta modificar la situación a su favor: el contrincante.

Cuando jugamos nosotros, en ocasiones razonamos de la siguiente forma: “si yo pongo aquí, el me pondrá aquí, yo podré mover esta, y así quizá le gane. Pero si yo pongo en este otro sitio, él lo hará aquí, y....”. Este método puede implementarse en un ordenador.

Para ello, el juego se ve como un conjunto de estados, en los que cada uno representa una posible situación del juego (o del tablero). Hay un estado inicial, que representa la situación de la partida antes de empezar a jugar, y, en principio, muchos estados finales en los que la partida ha terminado y algún jugador a ganado, o ha habido empate.

Para pasar de una situación del juego a otra, se efectúa una jugada, que se puede ver como aplicar un operador al estado en el que se está para ir a parar a un nuevo estado o situación del juego.

Para elegir un movimiento (operador a aplicar) lo que hacemos por lo tanto es una búsqueda en el conjunto de estados posibles, tratando de que la jugada que hagamos nos lleve a un estado final favorable para nosotros.

La diferencia con los problemas normales, es que no siempre somos nosotros quien aplicamos los operadores, por lo que no podemos decidir con exactitud el rumbo de la partida. Lo que sí podemos hacer es suponer que el contrario hará la misma jugada que haríamos nosotros en su situación, es decir que intentará ganar, o hacer la jugada que más nos perjudique a nosotros. Por lo tanto, en juegos en los que el turno va alternando, los operadores que se aplicarán serán, desde nuestro punto de vista, el mejor para nosotros, luego el peor (que será hecho por el contrario), luego el mejor, etc.

Si conseguimos modelizar en un ordenador los posibles estados del juego, y un algoritmo que realice esta particular búsqueda, podremos construir programas que juegan.

MINIMAX

Lo primero que se nos ocurriría es construir el árbol de posibles partidas entero evaluando cada nodo según lo beneficioso que sea para nosotros. Para ello, lo construimos recursivamente, de modo que los nodos finales sean los casos base, en los que la evaluación sea si ganamos nosotros, el contrario, o hay empate. Ese valor es recogido por el nodo antecesor. Si en la situación de la partida que modeliza nos tocaba mover a nosotros, moveremos a un nodo, si existe, que nos lleve a nosotros a la victoria, por lo que ese nodo se evaluará con la mejor evaluación posible de los nodos hijos. Si el turno era del contrario, moverá para originar un tablero que nos lleve a nosotros a la derrota, por lo que ese nodo se evaluará con la peor (para nosotros) evaluación posible de nodos hijos.

Por lo tanto, suponiendo turnos alternos, al ir bajando en el árbol de juego, los nodos del primer nivel elegirán el descendiente más prometedor (busca la máxima valoración de sus descendientes), los del segundo nivel escogerán al descendiente menos prometedor (busca la mínima valoración de sus descendientes), y así sucesivamente.

Esta búsqueda se denomina minimax, y tiene una implementación recursiva sencilla:

si fin de partida
    // Caso base.
    si ganamos nosotros **entonces** devolver 1
    si gana el contrario **entonces** devolver -1
    si hay empate **entonces** devolver 0
si no
    // Caso recursivo
    si es nuestro turno **entonces** devolver máxima evaluación de nodos hijos
    si no lo es **entonces** devolver mínima evaluación de nodos hijos.

Este algoritmo es suficiente para juegos simples, en los que el número de estados es pequeño. Sin embargo en los juegos habituales, el número de nodos es inmenso, y es una idea inocente pensar en construir todo el árbol. Hay que parar la búsqueda en algún momento, mucho antes de llegar a los estados finales. Esto tiene el problema de no saber, en los casos base, cuando se gana o pierde para darles una valoración que se propague. Habrá que construir una **función de evaluación** que nos diga lo bueno o malo de esa situación para nosotros. Ésta no nos devolverá solo tres posibles valores, si no un amplio abanico, de modo que cuanto mayor valor tenga, mejor será para nosotros, y cuanto peor valor, mejor para el contrario. Si la evaluación es positiva, la situación será ventajosa para nosotros, y si la evaluación es negativa, lo será para el jugador contrario.

Hablaremos más de las funciones de evaluación más adelante. El algoritmo general pasará ahora a ser:

si hemos profundizado suficiente
    // Caso base.
    devolver estimación del estado del juego
si no
    // Caso recursivo
    si es nuestro turno **entonces** devolver máxima evaluación de nodos hijos
    si no lo es **entonces** devolver mínima evaluación de nodos hijos.

Cuanto más profundicemos en la búsqueda, más cerca estaremos del final de la partida, y mejor será nuestra forma de jugar, pero más tiempo tardaremos en decidirnos.

Este algoritmo puede modificarse un poco, para evitarnos la distinción de casos en el caso recursivo. Podemos conseguir que ambos jugadores tengan que maximizar la evaluación de sus descendientes si en vez de evaluar a los nodos con lo bueno que son para nosotros lo evaluamos como lo bueno que son para el jugador que tiene el turno. Los casos base serán distintos: devolverán un 1 si el jugador que tiene el turno (que llamaremos A) gana, y un –1 si pierde. En el nodo antecesor tendrá, suponemos, el turno el jugador contrario (B). Si queremos que en los nodos recursivos siempre se maximice (para no tener que hacer unas veces el máximo y otras el mínimo), la evaluación que viene de abajo tendrá que ser la evaluación desde su punto de vista. Como no es el caso, para darles la vuelta, a la evaluación que le viene de sus hijos les cambia el signo. Así, si el jugador A ganaba (devolvía un 1), al cambiarle el signo en el nodo de B, la evaluación será –1 que, desde el punto de vista de B, es una mala situación, pues pierde.

Resumiendo, si queremos evitarnos la distinción de casos, los casos base se evalúan siempre en función del jugador que tenga el turno, y en los casos recursivo se escoge el máximo valor de los devueltos tras cambiarles el signo:

```
si hemos profundizado suficiente
        // Caso base.
        devolver estimación del estado del juego desde el punto de vista del
                jugador que tenga el turno.
si no
        // Caso recursivo
        devolver máximo(-evaluación de cada hijo)
```

Este es el minimax que hemos utilizado en la práctica.

PODA ALFA-BETA

Hemos dicho que el nivel de juego depende de la profundidad con la que busquemos en el espacio de soluciones. A más profundidad, más tiempo. Por lo tanto nos interesa desperdiciar cuanto menos tiempo mejor, para poder profundizar más en el espacio de soluciones, y poder jugar mejor.

Si analizamos más de cerca la forma de expansión de nodos del *minimax*, podemos darnos cuenta de que se podría ahorrar un número considerable de nodos (y de tiempo) a costa de un poco más de control de la expansión. Para explicarlo, nos olvidaremos de momento de la última modificación realizada sobre el minimax, en la que incluíamos los cambios de signo.

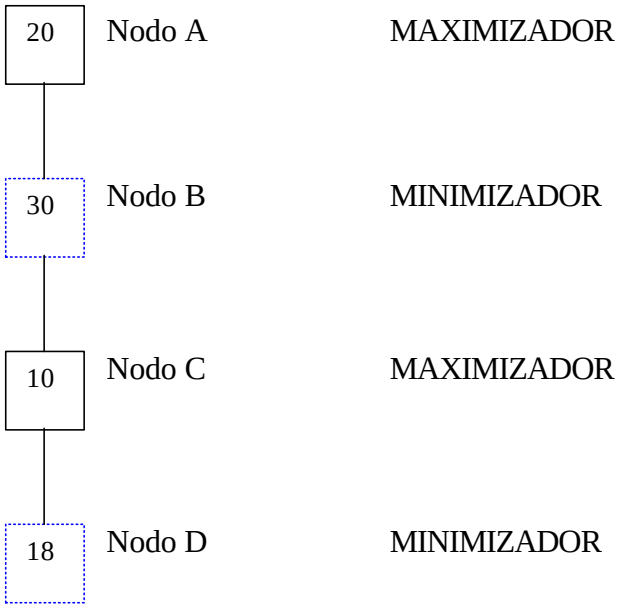
Pongamos el caso de que estamos en un nodo maximizador, que llamaremos A. Ese nodo ya ha evaluado al primer descendiente, y ha conseguido una evaluación de, por ejemplo, 20.

Manda expandir su segundo hijo (B), que será minimizador. Éste a su vez expande a su primer hijo (maximizador), que le devuelve una valoración de 10. Como el nodo B es minimizador, su evaluación será la mínima evaluación de todos sus nodos hijos. Por lo tanto, su evaluación será, como máximo, 10, pues cualquier valor mayor será desechado.

Por otro lado, el nodo A es maximizador, y ya tiene un valor de 20. Cualquier hijo que le devuelva una evaluación menor que eso no será tenido en cuenta. ¿Qué es lo que ocurre? Que el nodo B, como máximo, le va a devolver una valoración de 10, por lo que, antes de que B siga expandiendo a sus hijos, ya sabemos que B no nos vale para nada en la evaluación de A, y podríamos parar la búsqueda, y pasar a otro nodo de A.

Podemos hacer el mismo tipo de poda con nodos minimizadores que tienen hijos maximizadores con valores parciales mayores que el valor parcial de su predecesor.

Esto mismo se puede expandir a un número mayor de nodos maximizadores y minimizadores en expansión. Supongamos la siguiente situación en mitad de una expansión, en la que los cuatro nodos han expandido ya más de un hijo y tienen alguna evaluación parcial.



El nodo D no puede detener la expansión de sus hijos por el nodo C, pues la evaluación actual del nodo D es mayor que la del nodo C, y, por el momento, no podemos asegurar que el nodo C vaya a desechar su evaluación. De igual modo, no podemos detener la expansión de C por B, ni de B por A.

Sin embargo si miramos “más allá” desde D, podemos detener su expansión. ¿Por qué? El nodo D tiene una evaluación de 18, por lo que, como mucho, propagará hacia arriba un 18. Ese valor será inmediatamente cogido por el nodo C, y a su vez por el nodo B. Pero no por A, pues tiene un valor parcial mayor que el 18 que se propaga. ¿Para qué vamos a seguir expandiendo por lo tanto? Podemos detenernos ahí, y ahorrar algo de tiempo.

En definitiva, la poda alfa-beta va guardando dos valores durante la recursión, que se transmite a los nodos expandidos. El valor alfa es el máximo valor parcial conseguido en todos los nodos maximizadores que quedan por encima, y el valor beta el mínimo valor parcial conseguido en todos los nodos minimizadores que quedan por encima.

Si estamos en un nodo minimizador, y conseguimos un valor parcial menor que el valor alfa que nos han pasado desde arriba, el valor que pasemos hacia arriba va a ser desechado por algún nodo maximizador superior (el que tenga como valor parcial a alfa), y podemos parar la expansión.

Por su parte, si estamos en un nodo maximizador, y conseguimos un valor parcial mayor que el valor beta que nos han pasado, nuestra evaluación será desechada por algún nodo minimizador superior (el que tenga como valor parcial a beta), y podemos parar la expansión.

Cada paso de la recursión lleva ahora un poco más de tiempo, al tener que controlar los valores alfa y beta, pero el número de podas que se consiguen ahorran un tiempo suficiente como para compensar el tiempo perdido en el control.

```
si hemos profundizado suficiente
        // Caso base.
        devolver estimación del estado del juego
si no
        // Caso recursivo
        si es nuestro turno
                mejorValoracion = -infinito    // la peor posible
                para cada posible jugada
                        evaluación = evalua(nuevo nodo, alfa,beta)
                        si mejorValoracion peor que evaluación
                                mejorValoracion = evaluación
                        // Miramos si podemos podar.
                        si beta <= mejorValoracion
                                Podar; // Salimos del bucle.
                        sino
                                alfa = maximo (alfa, mejorVal);
        sino    // Turno contrario. Nodo minimizador
                mejorValoracion = +infinito   // la mejor posible
                para cada posible jugada
                        evaluación = evalua(nuevo nodo, alfa,beta)
                        si mejorValoracion mejor que evaluación
                                mejorValoracion = evaluación
                        // Miramos si podemos podar.
                        si alfa >= mejorValoracion
                                Podar; // Salimos del bucle.
                        sino
                                beta = mimimo (beta, mejorVal);
```

Igual que hemos hecho con el minimax, podemos aquí usar los signos para evitarnos la distinción de casos. Así, todos los nodos son maximizadores, y reciben de sus nodos hijos una evaluación a la que cambian el signo antes de considerarla. Al ser todos maximizadores, todos utilizan el valor beta para saber si tienen que podar o no, y actualizan el valor alfa para los nodos hijos.

En las llamadas recursivas, tenemos que convertir lo que ahora son nodos maximizadores en minimizadores, y al revés. Por eso cambiamos el signo al valor que se nos devuelve. Pero los valores de alfa y beta también dependen de los nodos maximizadores y minimizadores, por lo que también hay que intercambiarlos. Esto es debido a que en realidad, los nodos hijos deben utilizar al que ahora es el valor beta como valor alfa, por lo que donde la recursión espera recibir el valor alfa, nosotros le enviaremos nuestro valor beta, y viceversa.

Pero no es suficiente con esto. Nosotros tenemos en el valor beta el mínimo valor parcial de los que ahora son nodos minimizadores. Queremos convertirlo en el máximo de los que pasarán a ser nodos maximizadores, y para eso no basta con convertir al nuevo alfa en beta, sino que debemos, además, cambiarle el signo, para que un bajo valor de beta pase a ser un alto valor de alfa, difícil de superar.

Aunque la idea parece un poco enrevesada, al final el algoritmo queda muy sencillo:

```
si hemos profundizado suficiente
        // Caso base.
        devolver estimación del estado del juego para el jugador que
                tiene el turno
si no
        // Caso recursivo
        mejorValoracion = -infinito    // la peor posible
        para cada posible jugada
                evaluación = evalua(nuevo nodo, -beta,-alfa)
                si mejorValoracion peor que evaluación
                        mejorValoracion = evaluación
                // Miramos si podemos podar.
                si beta <= mejorValoracion
                        Podar; // Salimos del bucle.
                sino
                        alfa = maximo (alfa, mejorVal)
```

De nuevo, este es, más o menos, el algoritmo que hemos utilizado en la práctica.

2.- LA FUNCIÓN DE EVALUACIÓN

Una buena función de evaluación debe estimar lo cerca o lejos que una situación del juego está para llevar a un jugador concreto a la victoria. Además, es llamada muy a menudo durante la recursión, por lo que debe ser rápida, si queremos poder llegar a una profundidad razonable.

La cercanía o lejanía de un estado intermedio a un estado de victoria o de derrota, depende de multitud de factores, que varían con el tipo de juego. Cada uno de esos factores, tiene una importancia distinta, que puede variar incluso durante el desarrollo del juego. Habitualmente, las funciones de evaluación tendrán la forma:

e = factor1· característica1 + factor2· característica2 ...

Cada factor dará más o menos importancia a la característica que acompaña, y cada característica tendrá un mayor valor cuanto mejor se ajuste el tablero a la característica que modeliza.

Centremonos en la función de evaluación del Othello. Para ello, debemos buscar las características que deciden lo bueno o malo de una situación.

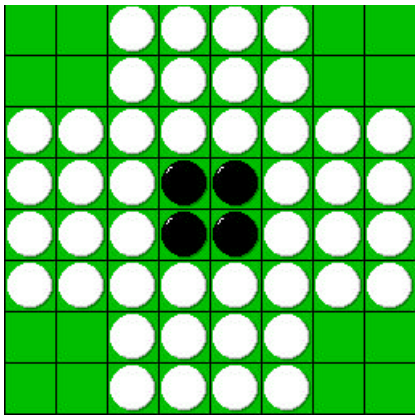
El objetivo del juego es bien conocido: conseguir más fichas que el oponente. Bueno, pues ya tenemos una. Cuantas más fichas tengamos, mejor será la situación para nosotros.

Esa caracterísitca es muy considerada por los jugadores noveles. Pero la realidad es que solo es buena en las fases finales del juego, cuando la profundidad nos lleva casi al final de la partida. Esto se debe a que cuantas más casillas nuestras tengamos... más puede comernos el contrario y más fácil es que cambien las cosas. Esto también ocurre al final, claro, pero si estamos lo suficiente cerca del final, no quedarán tantas jugadas donde nos pueda comer.

Por lo tanto, tenemos una característica, y hemos averiguado que su factor debe ser pequeño al principio, y mayor al final.

¿Qué factor es importante al principio de la partida? Para los jugadores expertos en Othello es importante la llamada **movilidad**. La movilidad es el número de casillas donde pueden poner. Cuantas más casillas tengamos donde poner, menos posibilidades existen de que el oponente nos obligue a poner en alguna casilla desafortunada porque no tengamos una opción mejor. Es decir, cuantas más casillas posibles tengamos, más posibilidades hay de que alguna no sea mala.

En ocasiones, la movilidad es más importante que el número de fichas incluso en etapas cercanas al final de la partida. Indirectamente, Joel Feinstein nos lo demuestra en un ejemplo, un tanto artificioso, que utiliza para otra cosa distinta.



Antes esta situación del tablero, podríamos pensar que las fichas blancas llevan las de ganar, pues tienen muchas más fichas, y estamos bastante cerca del final del juego. El turno es de negras.

La movilidad de blancas es totalmente nula, por lo que solo podrán poner donde les dejen las negras tras realizar su movimiento. Las negras pueden poner en tres de las cuatro casillas de cada esquina. Jugando con habilidad, negras deciden poner en la casilla X de cualquier esquina (una casilla X es una casilla adyacente a la esquina del tablero, en su misma diagonal). Las blancas no tienen más que una opción, comer en la esquina. Las negras pondrán en una casilla C (las adyacentes a una esquina , junto a un lateral), al lado de la blanca que se acaba de poner. Blancas pasan, negras ponen en la otra casilla C de esa esquina, blancas pasan, y se repite el asunto en otra esquina. Al final, negras ganan 46 a 18.

Esto demuestra que la movilidad es otro factor a tener en cuenta. Si miramos con más detalle lo que es la movilidad, llegaremos a la movilidad potencial, algo también apreciado por los jugadores humanos. Para que una casilla sea jugable para un color, debe tener alrededor alguna casilla del color contrario. Quizá en un momento dado en esa posición no se pueda poner, pero tener una ficha contraria al lado, puede originar que, en el futuro, esa casilla sea jugable.

La movilidad potencial es, por lo tanto, un tercer factor a incluir en nuestra función de evaluación aunque, evidentemente, tendrá menos valor que la movilidad real, pues al fin y al cabo cuenta con que en el futuro la casilla sea jugable.

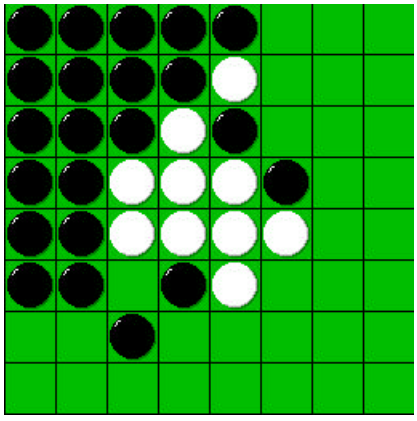
Para conseguir una buena movilidad, se deben tener muchas casillas libres donde poder poner, es decir muchas posibles fichas contrarias que poder comer. Por lo tanto, curiosamente, en las primeras etapas del juego, cuantas más casillas contrarias haya, más movilidad tendremos. Algunos jugadores humanos tratan de comer cuantas menos fichas contrarias mejor al principio, para tener más posibilidades en las siguientes jugadas.

Por último, algo muy interesante en una situación del tablero es el número de fichas estables que posee cada jugador. El número de fichas estables es el número de fichas que no pueden ser comidas o volteadas por el contrario de ninguna forma. Una ficha estable, por ejemplo, es cualquier esquina. Si una esquina contiene una ficha negra, esa ficha acabará siendo negra al final de la partida, pues para poderla comer tendría que ser rodeada por dos lados de una misma dirección de fichas blancas, y eso es imposible al estar en el límite del tablero. De igual forma, si un lateral tiene las ocho casillas con el mismo color, esas fichas serán estables, pues no hay forma de que sean rodeadas para comerlas.

Las fichas estables son importantes pues son fichas que cuentan al final como propias (o como contrarias) y carecen del defecto comentado antes para la característica del número de fichas del color, al no poder ser comidas.

Las casillas que más colaboran a conseguir fichas estables son las esquinas, pues son estables por sí mismas, no necesitan a ninguna otra para serlo. Por tanto son casillas a tener en cuenta.

Se pueden tener fichas estables no solo en las filas que forman el borde del tablero, sino también en las adyadentes. Esto se ve claramente en el tablero siguiente:



Las fichas negras tienen 18 fichas estables, apoyándose en los laterales superior e izquierdo. Además, en situaciones de este tipo, es fácil conseguir hacer pasar a las blancas, al tener poca movilidad potencial.

Aunque conocer el número real de fichas estables de cada color puede ayudar a la función de evaluación, calcular su valor es muy costoso, pues puede haber fichas estables en todo el tablero. Para ahorrar tiempo, los programas que juegan al Othello solo suelen buscar fichas estables en los laterales, que es donde aparecen más a menudo. Esto ahorra un tiempo apreciable en la función de evaluación, que permite mayores profundidades de búsqueda.

En nuestro programa hemos considerado esas cuatro características para evaluar un tablero. Algunos programas tienen en cuenta otras, como la accesibilidad a las partes del tablero (algo parecido a la movilidad, pero desde un punto de vista global), la paridad (tener en cuenta quien va a tener que empezar a jugar por la zona de las esquinas y que va a tener, posiblemente, que ceder esquinas a su contrario). Además, algunos jugadores humanos tratan de evitar jugar en los laterales, para no modificar fichas adyacentes a los laterales que pueden dar al contrario la oportunidad de conseguir laterales estables. Nada de esto lo hemos tenido en cuenta.

Algunos miran también el número de esquinas conseguidas. Nosotros no hemos metido eso en la función de evaluación explícitamente, pues puede incluirse algo de ese estilo en las tablas que valoran los laterales.

En resumen, la función de evaluación es:

```
eval = fichas· Factor_fichas[nº jugadas] +
     + movilidad· Factor_movilidad[nº jugadas] +
     + laterales· Factor_laterales[nº jugadas] +
     + movPotencial· Factor_movPotencial[nº jugadas]
```

donde:

```
fichas = fichas_jugador – fichas_contrario
movilidad = (movilidadAprox_jugador - movilidadAprox_contrario) /
            (movilidadAprox_jugador + movilidadAprox_contrario + 2)
laterales = evaluacionLaterales_jugador – evaluacionLaterales_contrario
movPotencial = (movPotencial_jugador - movPotencial_contrario) /
            (movPotencial_jugador + movPotencial_contrario + 2)
```

Para permitir la comparación entre distintas funciones de evaluación, en el programa se pueden cargar distintas “funciones de evaluación”, de la que se leen los coeficientes de cada característica en función del número de fichas puestas. Podemos así comparar funciones de evaluación que tienen en cuenta todas estas cosas, con la función de evaluación voraz que solo tiene en cuenta el número de fichas, o cualquier otra combinación. En el programa, se puede seleccionar el archivo de coeficientes en el nivel *Personalizado*.

3.- NUESTRA IMPLEMENTACIÓN DEL OTHELLO

Sabiendo todo lo anterior, implementar un Othello no parece demasiado complicado. Podemos modelizar el tablero con una matriz de 8*8 indicando si cada casilla está libre u ocupada y por qué color. Para realizar la recursión, podemos “fácilmente” saber en qué posiciones puede poner cada jugador. Para ello, se evalúa cada casilla vacía. Desde cada una de ellas recorremos el tablero en todas las direcciones posibles, mirando si hay una serie de fichas contrarias acabadas con una ficha propia. Si esto ocurre en alguna dirección, la posición será legal, y podremos poner ahí durante la recursión.

Por su parte, para saber el número de fichas, recorreremos el tablero incrementando el contador correspondiente según el color de cada casilla, para saber la movilidad lo volveremos a recorrer para saber en cuantos sitios puede poner cada color (igual que antes), para saber la movilidad potencial, lo recorremos mirando las fichas adyacentes a cada casilla vacía, y para las fichas estables, recorremos los laterales contando cuales no pueden ser comidas.

Implementar todo esto es muy fácil, pero también origina unos retrasos en la recursión increíbles, al tener que recorrer el tablero múltiples veces. Eso ocasiona una profundidad escasa, y el nivel de juego pasa a depender mucho más de la función de evaluación.

Hay que buscar técnicas que aceleren todos estos procesos que se repiten mucho durante la recursión. Para eso haremos uso de técnicas llamadas incrementales. Un ejemplo muy sencillo, pero válido, de este tipo de técnicas son la forma de contar las fichas. Podemos recorrer el tablero cada vez que queremos saber el número de casillas de cada color, y podemos también tenerlo en dos variables, al igual que tenemos la matriz con el tablero. Cada vez que se pone una nueva ficha, se incrementa el número de fichas de ese color, y cada vez que se da la vuelta a una, se decrementa el número de fichas del color inicial y se incrementa el del nuevo color. Este sencillo mecanismo nos ahorra recorrer el tablero en cada función de evaluación (que origina comparaciones en cada casilla para conocer el color), a costa de sumar y restar uno en los momentos que modifiquen el número de fichas de cada color. Y eso ahorra tiempo.

¿Podemos hacer algo así con el resto de recorridos? Evidentemente el número de fichas puede ser calculado incrementalmente muy fácilmente, pero no las posiciones donde se puede poner.

Pensemos en el recorrido que tenemos que hacer sobre los laterales para contar el número de fichas estables. Para cada lateral, tenemos que partir de una esquina hacia la otra, mirando si las fichas por las que pasamos son del mismo color que la esquina. Por cada ficha de igual color, una nueva ficha estable para ese color. Y luego otra vez para la esquina contraria. Y esto cuatro veces, una por cada lado. ¿Podemos mejorarlo? Cuando un lateral tiene una configuración establecida, el número de fichas estables no cambia. Podríamos por lo tanto contar el número de fichas estables solamente cuando se modifique el lateral, es decir cuando se ponga alguna nueva ficha en él.

Pero podemos mejorarlo más. Podríamos tener precalculadas el número de fichas estables para cada posible posición de un lateral. Tendríamos una tabla que dijera:

“Con todo vacío, 0 fichas estables para negras, y 0 para blancas”
“Con negra en esquina izquierda, 1 ficha estable para negras, y 0 para blancas”

y así para todas las posibles configuraciones. En tiempo de ejecución no tendríamos nada más que ir a la tabla con la situación actual, y leer cuantas fichas estables hay.

¿Es viable una tabla de este tipo? Primero, debemos saber su tamaño. Para ello, tenemos que averiguar cuantas posibles configuraciones distintas puede tener un lateral. Como hay 8 casillas, y cada casilla puede estar vacía, tener una ficha negra, o una blanca, hay 3^8 posibilidades, que son 6561. Una tabla con 6561 dobles entradas no es una tabla demasiado grande, si el tiempo que nos ahorra merece la pena.

El problema es cómo acceder a la tabla. Pero es sencillo: convertir la situación del lateral a un número en base 3. Así, la primera casilla tendrá peso 0 en el número, la siguiente tendrá peso 1, y así sucesivamente. Si una casilla está vacía, su dígito será un 0, si tiene una ficha negra su dígito será un 1, y si la ficha es blanca tendrá un 2. Por ejemplo:

y el valor del lateral será $1\cdot 3^0+1\cdot 3^1+2\cdot 3^2+2\cdot 3^3+2\cdot 3^4+2\cdot 3^5+0\cdot 3^6+0\cdot 3^7$, que es 724. A este valor le llamamos “valor resumido del lateral”.

Ya tenemos por lo tanto la forma de acceder a la tabla. La tabla guardará para cada índice (valor resumido de un lateral) el número de fichas estables de cada color. Para el 724, por lo tanto, guardará un 2 para negras, y un 0 para blancas.

Para evaluar los lados, hallaremos el valor resumido de cada lateral, y leeremos el valor en la tabla, sumando lo que necesitemos.

Pero... aquí falla algo. Resulta que no queremos recorrer los lados para ir más rápido. Y sin embargo tenemos que recorrerlos para hallar el valor resumido, y saber así qué índice usar en la tabla.

De nuevo podemos recurrir a las técnicas incrementales. Guardaremos, igual que el número de fichas y la matriz con tablero, el valor resumido de cada lado, que al principio será 0. Si se pone una nueva ficha en un lateral, su valor se incrementará según el color y la posición., y si se invierte una ficha del lateral, el valor resumido se incrementará o decrementará según el tipo de cambio de color y la posición de la ficha. Para acelerarlo aún más, tendremos una matriz con las potencias de 3 precalculadas.

Con esto, de nuevo complicamos los procedimientos de poner ficha e invertir ficha al tener que actualizar variables nuevas. Pero esta actualización es mucho más rápida en comparación con los recorridos del tablero en las funciones de evaluación, por lo que, de nuevo, ganamos velocidad.

Pero resulta que la tabla solo se utiliza en la función de evaluación. Y la función de evaluación no tiene en cuenta el número de fichas estables del color actual, sino la resta del número de fichas estables del color actual, y del contrario. Por lo tanto, en la función de evaluación siempre se accede a la tabla dos veces, y se hace la resta. Podemos ahorrarnoslo, si guardamos directamente la resta. Así, en la tabla guardaremos el número de fichas estables de negras menos el de blancas. Si resulta que el turno era de negras, se utiliza el valor de la tabla. Y si el turno es de blancas, se le cambia el signo antes de usarlo. Esto no se ha hecho con la idea de ahorrar tiempo, sino para ahorrar espacio en la tabla.

Por otro lado, si construimos el programa para que la tabla de vértices se obtenga de un fichero externo, podemos ganar además versatilidad. Igual que podíamos cargar distintos coeficientes para construir nuevas funciones de evaluación sin necesidad de modificar el código, ahora podemos tener distintas tablas con evaluaciones de los laterales según distintos criterios, para compararlas. En nuestro programa, al igual que ocurría con los coeficientes de la función de evaluación, podemos escoger una tabla en el nivel de juego *Personalizado*.

Así, además de la tabla con el número de fichas estables, tenemos otra con el número medio de fichas que se consiguen desde cada configuración. Es decir, para cada configuración del lateral, se han construido todas las partidas sobre un tablero unidimensional de Othello, llegando así a todas los posibles estados finales alcanzables desde cada configuración. Luego, se calcula la media de fichas que se obtienen en todas esas configuraciones finales, y se le asigna ese valor a la configuración inicial. Una partida unidimensional sería una partida “normal” sobre una única fila, en la que el turno pasa alternativamente, y cada jugador puede poner en cualquier casilla (podría poner porque come alguna ficha de una fila distinta en diagonal) o pasar (porque pone ficha en cualquier otra parte del tablero).

Por el momento hemos mejorado la velocidad para averiguar el número de fichas de cada color, y para la evaluación de los laterales. Nos quedan tres recorridos: el que hacemos para averiguar la movilidad, el que hacemos para hallar la movilidad potencial, y el que hacemos para buscar las casillas donde se puede poner en los nodos no terminales de la recursión. ¿Podemos mejorarlos y ahorrarnos tiempo?.

Dos de esos recorridos son en realidad comunes. Cuando recorremos el tablero buscando donde puede poner un jugador para poder seguir la recursión, comprobamos cada casilla vacía para saber si el jugador puede poner. Si es así, lo incluimos en la lista de posiciones legales y continuamos la búsqueda. Cuando deseamos saber la movilidad, debemos hacer lo mismo, pero cada vez que encontramos una casilla legal, incrementamos el contador de la movilidad.

Este recorrido del tablero atraviesa todas las casillas, igual que hacía el que contaba el número de fichas. Pero éste es distinto. Para cada casilla vacía, hay que hacer a su vez otros 8 recorridos, uno por sentido en el que se puede comer, buscando una línea de fichas contrarias acabadas por una ficha nuestra. Recorrer las 64 casillas del tablero es lento, pero aún lo es más si por cada una de ellas tenemos que recorrer otra vez el tablero en 8 direcciones distintas.

Lo primero que se nos ocurre es tener otra tabla, indicando qué casillas son legales y cuales no para cada jugador. Así nos evitamos los recorridos auxiliares en cada casilla, y será suficiente con recorrer el tablero una vez. Además, podríamos tener un contador que indicara el número de casillas donde puede poner cada color. Esta técnica la utiliza Olivier Arsac y sus compañeros en su juego del Othello llamado Darwersi.

Como ventaja tiene que para la función de evaluación, calcular la movilidad de cada jugador es instantáneo, y para saber donde se puede poner hay que hacer un recorrido simple por una tabla. El problema es que se incrementa enormemente la complejidad de los procedimientos que ponen una ficha o que la invierten, pues es en ellas en las que se hacen los recorridos que nos evitamos en los otros dos lados. Así, cada vez que se pone una nueva ficha hay que recorrer el tablero en todas las direcciones para modificar la movilidad de las casillas que se vean alteradas por la nueva ficha. Y cuando se invierte una ficha, hay que hacer también esos recorridos,

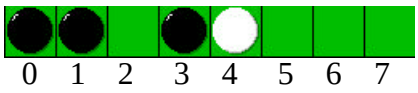
pues quizá la ficha anterior facilitaba la movilidad en algunas casillas que ahora pasan a no ser legales, o la nueva ficha permite comer en otras. En un estudio que ellos mismos realizan, el procedimiento de poner una ficha se lleva el 38% del tiempo de la recursión, debido a su mayor complejidad.

Hay otra técnica, que acelera los procedimientos de poner y cambiar una ficha, a costa de no permitir calcular la movilidad con la simple consulta de una variable.

Lo que queremos evitar son los recorridos partiendo de una posición en todas las direcciones posibles, para saber si se puede o no comer. En los laterales, buscabamos el número de fichas estables, y ahora en estos recorridos que queremos quitar buscamos una serie de fichas del color contrario acabados en una ficha de nuestro color. No deja de ser un recorrido, que también podemos tener precalculado. Tendremos una tabla precalculada, en la que cada índice indica el valor resumido de una fila. La tabla almacenará dos valores de 16 bits, uno por cada color.

Cada valor se divide en 8 grupos de 2 bits, cada grupo para una casilla de la fila. El primer bit indicará si puede comer hacia la izquierda (por ejemplo), y el segundo hacia la derecha.

Por ejemplo, ante la fila siguiente:



las fichas negras solo pueden poner en la casilla 5, y comen hacia la izquierda. Por lo tanto la entrada de negras en la tabla tendrán todos los bits a 0, salvo el bit de sentido izquierdo de la casilla 5.

Por su parte, las blancas pueden poner en la casilla 2, por lo que la entrada de blancas tendrán todos los bits a 0 salvo el de sentido derecho de la casilla 2.

Para saber si un color puede poner en una casilla determinada, consultaremos la tabla según el valor resumido, miraremos los bits de la posición que nos ocupa (con una máscara) y si el valor obtenido no es 0, la casilla será legal.

Pero para que una casilla sea legal, hay que poder comer en horizontal, vertical y diagonal. Por lo tanto tendremos que mirar en la tabla usando los valores resumidos de la columna en la que esté la casilla, de su fila y de sus dos diagonales. Si alguno no está a cero, la casilla será legal. Para que todo esto sea lo suficientemente rápido, tendremos que guardar los valores resumidos, al igual que hacíamos con los laterales, de todas las filas, columnas y diagonales del tablero. Y actualizarlas convenientemente, con cada nueva ficha puesta, o ficha invertida.

Antes de continuar, veamos como guardar todos los valores resumidos. Tenemos que administrar los valores resumidos de 8 filas, de 8 columnas, de 15 diagonales ‘/’ y de otras 15 diagonales ‘\’.

Para las columnas es sencillo. Tenemos una matriz de ColumnaAbreviada[1..8], y accederemos al valor resumido de la columna en la que está situada una ficha utilizando su coordenada X. Luego para saber qué bits del valor leído se corresponden con la casilla que nos trata, usaremos la coordenada Y.

Para las filas es semejante. Tendremos la matriz FilaAbreviada[1..8], accederemos a ella utilizando la coordenada Y de la casilla, y escogeremos los bits correctos utilizando la coordenada X.

Pero para las diagonales el asunto es más complicado. Tenemos dos matrices de diagonales, Diagonal1Abreviada [0..14] para las diagonales ‘\’, y para las ‘/’ Diagonal2Abreviada[0..14]. Hay que encontrar una fórmula que, en función de las coordenadas X e Y nos dé la misma diagonal para las casillas que pertenezcan a ella.

Nosotros hemos decidido usar las fórmulas siguientes:

7 – x + y para las diagonales ‘\’

x + y – 2 para las diagonales ‘/’

que numeran las diagonales del tipo ‘\’ desde la esquina superior derecha hacia abajo a la izquierda, y las del tipo ‘/’ desde la esquina superior izquierda hacia abajo a la derecha. Para acceder a los bits de cada casilla, hemos decidido utilizar la coordenada Y de la casilla en ambos casos.

Continuemos. ¿Es rentable utilizar ésta técnica? Ahora tenemos que complicar aún más los procedimientos que ponen e invierten fichas, aunque no tanto como con la técnica implementada en el Darwersi. Sin embargo, para saber dónde se puede poner, en vez de tener que hacer 8 recorridos, hacemos un acceso a cuatro tablas distintas.

Pero además se puede rentabilizar ese coste con otra utilidad más.

Supongamos que decidimos poner en una posición, que sabemos, por tablas o por recorridos, que es legal. Al poner la ficha, tendremos que realizar los 8 recorridos para saber en qué sentido comemos. Iremos así hacia la derecha buscando una serie de fichas contrarias acabadas con una ficha nuestra. Si lo encontramos, repetiremos el recorrido dando la vuelta a las fichas contrarias. Y eso para los 8 sentidos posibles.

Teniendo las tablas, podemos saber en qué sentidos comemos, por lo que no necesitaremos recorridos de comprobación. Averiguamos los sentidos correctos consultando los bits correspondientes, y nos ponemos a dar la vuelta a fichas contrarias sin más comprobaciones.

La tabla por lo tanto nos ahorra tiempo en tres ocasiones: cuando comemos una ficha, cuando queremos saber las posiciones legales, y cuando queremos hallar la movilidad. Queremos hacer las tres cosas un gran número de veces durante la recusión, por lo que el tiempo extra que perdemos en actualizar los valores resumidos de columnas, filas y diagonales es compensado por el tiempo que nos ahorramos usándolos. La técnica incremental vuelve a ser una buena solución.

Pero si la primera técnica vista para la movilidad conseguía hallarla sin recorridos, ¿podríamos hacer nosotros algo parecido ahora? Si en la función de evaluación nos conformamos con una aproximación de la movilidad, podemos, en efecto, hacer algo.

Si en vez de utilizar el número de casillas legales, utilizamos el número de direcciones en los que cada jugador puede comer, podemos calcularlo de forma incremental. En efecto, en la tabla precalculada con las posiciones donde podemos comer podríamos guardar un nuevo valor, que indique el número de casillas donde cada color puede comer para una configuración dada. Podemos tener dos variables, igual que las teníamos para el número de fichas, que indiquen la suma de esas entradas para todas los valores resumidos de filas, columnas y diagonales. Cuando ponemos una nueva ficha, se modifica la variable restando las casillas legales que nos proporcionaban la

fila, columna y diagonales de esa posición antes de poner la ficha, y sumando las que nos proporcionan tras poner la nueva ficha.

En realidad no es tan sencillo, debido a las diagonales. En las diagonales que no son las principales del tablero, hay partes del valor resumido que no corresponden ninguna casilla, debido a que la diagonal tenga menos de 8 casillas. En algunas ocasiones podría comerse en diagonal una ficha en un borde si se pudiera poner “fuera” del tablero. El programa nunca mira, lógicamente, en esas posiciones al calcular la movilidad. Pero cuando accedemos a la tabla que nos dice la movilidad aproximada de esa diagonal, cometeremos un error pues la tabla sí considera como legal esa posición, y la contaríamos. Hay que meter por lo tanto una pequeña corrección en las diagonales, que puede hacerse rápidamente utilizando una tabla auxiliar de máscaras para saber si estamos cayendo en el error, y rectificarlo.

El caso es que con todo esto conseguiremos tener una aproximación de la movilidad. No es la movilidad real, pues si en una casilla podemos comer en dos direcciones diferentes (hacia arriba y hacia la derecha, por ejemplo), la movilidad aproximada que estamos guardando considerará dos veces a esa posición, en vez de una sola como haría la movilidad real. Sin embargo, si pudiera comer hacia la izquierda, y hacia la derecha, solo sería contado una vez, pues come en una sola dirección, y en la tabla de la fila de esa ficha solo se cuenta una vez.

Antes de entrar a valorar si la aproximación de la movilidad nos sirve para la función de evaluación, veamos otra de sus utilidades. Durante la expansión de un nodo en la recursión, es habitual preguntar si un jugador puede o no poner ficha, es decir si pasa. Si esto ocurre, no se realiza jugada alguna, y el turno pasa al jugador contrario. Saber si un jugador pasa o no, ocasionaría conocer la movilidad, es decir si un jugador puede o no poner en alguna casilla. Esa movilidad no se necesita para la expansión del nodo (solo se usa en la función de evaluación), por lo que estamos haciendo un recorrido extra. Para evitarnoslo, podemos utilizar la movilidad aproximada. Ésta, recordemoslo, indicaba el número de direcciones en las que un jugador podía comer en una configuración del tablero determinada. Si no hay ninguna dirección en la que comer, el jugador pasa. Por lo tanto no tenemos que recorrer el tablero para averiguar si debemos pasar el turno al contrario directamente o no.

¿Nos vale además para la función de evaluación? El mejor programa de Othello del mundo, el Logistello, la utiliza en su función de evaluación, por lo que si puede sustituirse. El problema es que para poder utilizar la aproximación, necesitamos una función de evaluación lo suficientemente buena como para poder “ocultar” el error que comete, o usar una profundidad suficiente como para que deje de tener importancia. Logistello llega a la increíble profundidad de 16, por lo que puede permitirse el lujo de utilizarla.

¿Qué hacemos nosotros? Bueno, nuestro programa no llega, ni de lejos, a esas velocidades. Si utilizamos la aproximación, no juega tan bien como si utilizamos la movilidad real. Pero utilizando la movilidad real, el nivel de profundidad permitido disminuye. Por lo tanto, vamos a utilizar la movilidad aproximada.

Tal vez podríamos haber mezclado las dos técnicas (la que usa el Darwersi y la de la tabla precalculada) para poder tener la movilidad real en variables sin tener que gastar tanto tiempo en poner fichas como le ocurre al Darwersi. Desgraciadamente,

descubrimos la técnica del Darwersi demasiado tarde, como para plantearnos cambiarlo. Quizá en una segunda versión del juego podríamos mezclarlos...

El único recorrido que nos falta por mejorar es el de la movilidad potencial. Con todo lo que sabemos ya de técnicas incrementales no debería ser difícil.

En realidad la definición de movilidad potencial no es única. Pueden entenderse tres cosas distintas:

-Número de fichas del color en cuestión que tienen alrededor alguna casilla vacía.

-Número de casillas vacías que tienen al menos una casilla del color alrededor.

-Suma de casillas vacías adyacentes a las fichas del color que nos ocupa.

Queremos hallarlo de forma incremental, de modo que la que mejor se adapta a nuestras necesidades es la tercera. De ese modo, cada vez que pongamos una nueva ficha, sumamos el número de casillas vacías alrededor a la movilidad del jugador contrario, pues ahora hay una ficha más nuestra que a él le beneficia. Y cuando damos la vuelta a una ficha, el número de casillas vacías alrededor se decrementa de la movilidad del jugador que consigue la ficha (pierde movilidad al perder una ficha el contrario), y se lo sumamos a la movilidad potencial del jugador que la pierde (su contrario gana una ficha, y su movilidad crece).

Para no tener que recorrer las casillas adyacentes a una dada para saber cuantas casillas vacías hay, tenemos una matriz de “libertades”. Ese término es muy usado en el juego del Go, e indica el número de casillas vacías alrededor de una dada. Podemos usar este valor directamente, que habrá que modificar incrementalmente cada vez que se ponga una ficha.

Resumiendo, para mejorar la velocidad de la función de evaluación hemos complicado más las funciones de poner una nueva ficha e invertir una ficha, al tener que actualizar variables que luego consultaremos a menudo.

Durante la recursión, se hacen copias del tablero original y se modifica, realizarndo jugadas sobre ellas. Lo habitual en la programación orientada a objetos sería pedir memoria para cada tablero auxiliar que se requiere, y liberarla cuando ya no se necesite. Sin embargo la gestión de la memoria es muy lenta, sobre todo con el número de veces que debemos pedirla y liberarla en la recursión. Por ello, en el programa hemos utilizado una “pila de tableros” que es estática, de modo que se van utilizando los tableros guardados en ella, y no se necesita llamar al gestor de memoria del sistema operativo. Esta es la última técnica utilizada para acelerar el programa.

4.- CÓDIGO MÁS SIGNIFICATIVO

El programa se ha implementado en Visual C++, con programación orientada a objetos.

Las dos clases más importantes son la clase Tablero, y la clase Pensador. La primera se encarga de almacenar todas las variables que describen el estado de un tablero, es decir la matriz con las fichas y todas las requeridas por las técnicas incrementales. De esta clase, los métodos más importantes son los que ponen una ficha, los que invierten una ficha, y los que miran si se puede o no poner en una posición determinada.

```
void Tablero :: PonFicha (int x, int y, int color) {
// Pone una ficha (sin comer ni nada de eso) del color especificado
// en la posición (x,y), y actualiza los atributos del objeto.
// Se ha hecho en un método a parte por la complicación añadida de la
// técnica incremental.
// Como el método es privado, se supone que la casilla (x,y) estará
// vacía, y es el método del objeto que llama a este el que se encarga
// de asegurar eso.

  int contrario;
  int contX, contY;
  contrario = colorContrario(color);

        // Modificamos el número de fichas del color.
        numFichas[color]++;
        numFichas[0]++;

        // Modificamos la movilidad aproximada. Para ello primero restamos la
        // debida a la columna, fila y diagonales que ahora se modifican.
        movilidadAproximada[0] -=
                TablaComer[columnaAbreviada[x-1]].numPos[0] +
                TablaComer[filaAbreviada[y-1]].numPos[0] +
                TablaComer[diagonal1Abreviada[7-x+y]].numPos[0] +
                TablaComer[diagonal2Abreviada[x+y-2]].numPos[0];
        // Hacemos la rectificación por las diagonales (ver declaración de la
        // matriz ANDDiagonal, en este mismo archivo).
        if (TablaComer[diagonal1Abreviada[7-x+y]].posiciones[0] & ANDDiagonal[7-x+y])
                movilidadAproximada[0]++;
        if (TablaComer[diagonal2Abreviada[x+y-2]].posiciones[0] & ANDDiagonal[x+y-2])
                movilidadAproximada[0]++;

        movilidadAproximada[1] -=
                TablaComer[columnaAbreviada[x-1]].numPos[1] +
                TablaComer[filaAbreviada[y-1]].numPos[1] +
                TablaComer[diagonal1Abreviada[7-x+y]].numPos[1] +
                TablaComer[diagonal2Abreviada[x+y-2]].numPos[1];
        if (TablaComer[diagonal1Abreviada[7-x+y]].posiciones[1] & ANDDiagonal[7-x+y])
                movilidadAproximada[1]++;
        if (TablaComer[diagonal2Abreviada[x+y-2]].posiciones[1] & ANDDiagonal[x+y-2])
                movilidadAproximada[1]++;

        // Modificamos el valor de los índices (valores resumidos)
        // Para eso, sumamos el dígito de la ficha en el peso correcto.
        columnaAbreviada[x-1] += color*Potencia3[y-1];
        filaAbreviada[y-1] += color*Potencia3[x-1];
        diagonal1Abreviada[7-x+y] += color*Potencia3[y-1];
        diagonal2Abreviada[x+y-2] += color*Potencia3[y-1];
```

```
        // Terminamos de modificamos la movilidad aproximada. Sumamos la
        // debida a la columna, fila y diagonales que ya se han modificado.
        movilidadAproximada[0] +=
                TablaComer[columnaAbreviada[x-1]].numPos[0] +
                TablaComer[filaAbreviada[y-1]].numPos[0] +
                TablaComer[diagonal1Abreviada[7-x+y]].numPos[0] +
                TablaComer[diagonal2Abreviada[x+y-2]].numPos[0];
        if (TablaComer[diagonal1Abreviada[7-x+y]].posiciones[0] & ANDDiagonal[7-x+y])
                movilidadAproximada[0]--;
        if (TablaComer[diagonal2Abreviada[x+y-2]].posiciones[0] & ANDDiagonal[x+y-2])
                movilidadAproximada[0]--;

        movilidadAproximada[1] +=
                TablaComer[columnaAbreviada[x-1]].numPos[1] +
                TablaComer[filaAbreviada[y-1]].numPos[1] +
                TablaComer[diagonal1Abreviada[7-x+y]].numPos[1] +
                TablaComer[diagonal2Abreviada[x+y-2]].numPos[1];
        if (TablaComer[diagonal1Abreviada[7-x+y]].posiciones[1] & ANDDiagonal[7-x+y])
                movilidadAproximada[1]--;
        if (TablaComer[diagonal2Abreviada[x+y-2]].posiciones[1] & ANDDiagonal[x+y-2])
                movilidadAproximada[1]--;

        // Decrementamos las libertades de las casillas adyacentes en 1.
        for (contX = -1; contX <=1; contX++)
        for (contY = -1; contY <=1; contY++)
                libertades[x+contX][y+contY]--;
        // Con estos dos for, también decrementamos la libertad de (x,y). La
        // restauramos. (así nos evitamos un if en el bucle, que siempre
        // viene bien...)
        libertades[x][y]++;

        // Actualizamos (incrementamos) la movilidad potencial del contrario.
        movilidadPotencial[contrario-1] += libertades[x][y];

        // Ponemos la ficha en el tablero.
        tablero[x][y] = color;

} // PonFicha
```

void Tablero :: Invierte (int x, int y) {

```
// Da la vuelta a la ficha (x,y) y actualiza los atributos del
// objeto.
// Se ha puesto en un procedimiento separado por la complicación añadida
// de las técnicas incrementales.
// Como el método es privado, se supone que la casilla (x,y) estará
// ocupada, y es el método del objeto que llama a este el que se encarga
// de asegurar eso.

        int color, contrario;
        int sumarRestar;

        // contrario es el color que pierde la ficha, y color el que la gana.
        contrario = tablero[x][y];
        color = colorContrario(contrario);
```

```
        // Modificamos el número de fichas de cada color.
        numFichas[color]++;             // Una más del color...
        numFichas[contrario]--;         // ...y una menos del contrario.

        // Modificamos la movilidad aproximada. Para ello primero restamos la
        // debida a la columna, fila y diagonales que ahora se modifican.
        movilidadAproximada[0] -=
                TablaComer[columnaAbreviada[x-1]].numPos[0] +
                TablaComer[filaAbreviada[y-1]].numPos[0] +
                TablaComer[diagonal1Abreviada[7-x+y]].numPos[0] +
                TablaComer[diagonal2Abreviada[x+y-2]].numPos[0];
        if (TablaComer[diagonal1Abreviada[7-x+y]].posiciones[0] & ANDDiagonal[7-x+y])
                movilidadAproximada[0]++;
        if (TablaComer[diagonal2Abreviada[x+y-2]].posiciones[0] & ANDDiagonal[x+y-2])
                movilidadAproximada[0]++;

        movilidadAproximada[1] -=
                TablaComer[columnaAbreviada[x-1]].numPos[1] +
                TablaComer[filaAbreviada[y-1]].numPos[1] +
                TablaComer[diagonal1Abreviada[7-x+y]].numPos[1] +
                TablaComer[diagonal2Abreviada[x+y-2]].numPos[1];
        if (TablaComer[diagonal1Abreviada[7-x+y]].posiciones[1] & ANDDiagonal[7-x+y])
                movilidadAproximada[1]++;
        if (TablaComer[diagonal2Abreviada[x+y-2]].posiciones[1] & ANDDiagonal[x+y-2])
                movilidadAproximada[1]++;

        // Modificamos el valor de los índices (valores resumidos)
        // Para eso, sumamos o restamos un 1 en el peso correcto.
        // Basta con sumar o restar, pues en ese dígito ya había un 1 o un 2,
        // y ahora lo queremos pasar a 2 o 1.
        sumarRestar = color - contrario;
        // sumarRestar tendrá un 1 si la ficha pasa a ser BLANCA (el dígito pasa
        // de 1 a 2), y -1 si la ficha pasa a ser NEGRA (el dígito pasa de
        // 2 a 1).
        columnaAbreviada[x-1] += sumarRestar*Potencia3[y-1];
        filaAbreviada[y-1] += sumarRestar*Potencia3[x-1];
        diagonal1Abreviada[7-x+y] += sumarRestar*Potencia3[y-1];
        diagonal2Abreviada[x+y-2] += sumarRestar*Potencia3[y-1];

        // Terminamos de modificamos la movilidad aproximada. Sumamos la
        // debida a la columna, fila y diagonales que ya se han modificado.
        movilidadAproximada[0] +=
                TablaComer[columnaAbreviada[x-1]].numPos[0] +
                TablaComer[filaAbreviada[y-1]].numPos[0] +
                TablaComer[diagonal1Abreviada[7-x+y]].numPos[0] +
                TablaComer[diagonal2Abreviada[x+y-2]].numPos[0];
        if (TablaComer[diagonal1Abreviada[7-x+y]].posiciones[0] & ANDDiagonal[7-x+y])
                movilidadAproximada[0]--;
        if (TablaComer[diagonal2Abreviada[x+y-2]].posiciones[0] & ANDDiagonal[x+y-2])
                movilidadAproximada[0]--;

movilidadAproximada[1] +=
        TablaComer[columnaAbreviada[x-1]].numPos[1] +
  TablaComer[filaAbreviada[y-1]].numPos[1] +
  TablaComer[diagonal1Abreviada[7-x+y]].numPos[1] +
  TablaComer[diagonal2Abreviada[x+y-2]].numPos[1];
        if (TablaComer[diagonal1Abreviada[7-x+y]].posiciones[1] & ANDDiagonal[7-x+y])
                movilidadAproximada[1]--;
        if (TablaComer[diagonal2Abreviada[x+y-2]].posiciones[1] & ANDDiagonal[x+y-2])
                movilidadAproximada[1]--;
```

```
        // Modificamos las movilidades potenciales de los dos jugadores.
        movilidadPotencial[contrario-1] += libertades[x][y];
        movilidadPotencial[color-1] -= libertades[x][y];

        // Damos la vuelta a la ficha...
        tablero[x][y] = color;

} // Invierte
```

unsigned char Tablero :: PuedePoner (int x, int y, int color) {

```
// Devuelve si se puede poner o no en la casilla (x,y).
// Si devuelve 0, no se puede poner. Si devuelve un valor distinto,
// éste se considera un campo de bits. Los bits 0 y 1 indican si se puede
// comer en esa columna, los bits 2 y 3 en esa fila, los bits 4 y 5 en
// la diagonal \ y los bits 6 y 7 en la diagonal /. Para cada dirección
// hay dos bits, uno por sentido. El bit menos significativo de cada caso
// indican el sentido de menos significativo a más significativo (de abajo
// a arriba en columnas y diagonales, y de izquierda a derecha en las
// filas).
// Ocupa mucho, pero casi todo son comentarios. Esta función es muy rápida.

        unsigned char devolver;
        WORD campoActual;

        if (libertades[x][y] == LibertadInicial[x][y])
                // Si la casilla no tiene ninguna ficha alrededor, no se podrá
                // poner en ella. No tendrá ninguna ficha alrededor si la libertad
                // de la casilla es igual que la original. Además con este caso
                // se incluye que la casilla sea vacía. Si no está vacía, es porque
                // se ha colocado en ella antes una ficha, y para eso, ha tenido
                // que comer alguna ficha, por lo que la libertad de la casilla al
                // poner en ella ya no era igual que al principio. Esto es correcto
                // también para las 4 fichas iniciales.
                return (0);

        // En principio, se puede poner la ficha. Pero hay que ver si poniendo
        // ahí la ficha se come alguna en cualquier dirección.

        campoActual = TablaComer[columnaAbreviada[x-1]].posiciones[color-1];
        // campoActual contiene el campo de posiciones donde puede poner
        // el color especificado. Vemos si en la posición en cuestión puede
        // poner. Rotamos para que los bits de la casilla quede en los bits
        // menos significativos. Luego borramos el resto de bits.
        campoActual >>= 2*(y-1);
        campoActual &= 3;
        // Si los bits menos significativos no están a 0, podrá comer, y si
        // está a 0 no podrá. Justo ese es el significado de los 2 bits menos
        // significativos del valor que debemos devolver. Los guardamos.
        devolver = campoActual;

        campoActual = TablaComer[filaAbreviada[y-1]].posiciones[color-1];
        // campoActual contiene el campo de posiciones donde puede poner
        // el color especificado. Vemos si en la posición en cuestión puede
        // poner. Para eso rotamos para que los bits de la casilla quede en los
        // bits 2 y 3. Luego borramos el resto de bits.
        campoActual >>= 2*(x-1);        // Lo llevamos al bit menos significativo.
        campoActual &= 3;               // Borramos el resto de bits.
```

```
        campoActual <<= 2;              // Lo llevamos a los bits 2 y 3.
                                        // (esto nos ahorra una distinción de casos).
        // Tenemos en los bits 2 y 3 si puede comer en la
        // columna. Justo eso es lo que se necesita en los bits 2 y 3 del
        // valor devuelto.
        devolver |= campoActual;


        campoActual = TablaComer[diagonal1Abreviada[7-x+y]].posiciones[color-1];
        // campoActual contiene el campo de posiciones en \ donde puede poner
        // el color especificado. Vemos si en la posición en cuestión puede
        // poner. Para eso rotamos para que los bits de la casilla quede en los
        // bits 4 y 5. Luego borramos el resto de bits.
        campoActual >>= 2*(y-1);        // Lo llevamos al bit menos significativo.
        campoActual &= 3;               // Borramos el resto de bits.
        campoActual <<= 4;              // Lo llevamos a los bits 4 y 5.
                                        // (esto nos ahorra una distinción de casos).
        // Tenemos en los bits 4 y 5 si puede comer en la
        // columna. Justo eso es lo que se necesita en esos bits del
        // valor devuelto.
        devolver |= campoActual;

        campoActual = TablaComer[diagonal2Abreviada[x+y-2]].posiciones[color-1];
        // campoActual contiene el campo de posiciones en / donde puede poner
        // el color especificado. Vemos si en la posición en cuestión puede
        // poner. Para eso rotamos para que los bit de la casilla queden en los
        // bits 6 y 7. Luego borramos el resto de bits.
        campoActual >>= 2*(y-1);        // Lo llevamos al bit menos significativo.
        campoActual &= 3;               // Borramos el resto de bits.
        campoActual <<= 6;              // Lo llevamos al cuarto bit menos signif.
                                        // (esto nos ahorra una distinción de casos).
        // Tenemos en los bits 6 y 7 si puede comer en la
        // diagonal. Justo eso es lo que se necesita en el cuarto bit del
        // valor devuelto.
        devolver |= campoActual;

        // Ya hemos mirado en todas las direcciones. Acabamos.
        return (devolver);
}
```

La clase pensador implementa los métodos que deciden qué jugada realizar. Posee métodos para indicarle qué ficheros utilizar para la función de evaluación y para los laterales.

El método “Pensar” devuelve la mejor posición para la función de evaluación y profundidad establecidas. Se apoya en los métodos auxiliares “Minimax” y “Alfabeta”, que realizan la búsqueda propiamente dicha. El método Pensar es por lo tanto el método que prepara las llamadas a los procedimientos de búsqueda. Esto nos permite comprobar antes de la búsqueda si realmente el jugador que tiene el turno puede poner, o si solo tiene una posibilidad para no hacer búsqueda y poner directamente en la única casilla legal.

```
Posicion Pensador :: Pensar(Tablero tab, int color) {
// Devuelve la posición donde el objeto considera que es mejor
// poner en el tablero.
// El minimax y alfabeta solo devuelven las valoraciones de los tableros,
// pero no la posición donde se pondría, para ahorrar tiempo y quitarse
// un parámetro que solo es útil en la primera llamada. En esta función
// se realiza por lo tanto una búsqueda de la mejor posición, evaluando
// todos los posibles movimientos para el tablero actual, y llamando a
// minimax o alfabeta según proceda. Esto nos permite comprobar al prin-
// cipio algunos movimientos especiales. Miramos así el número de movimien-
// tos posibles, de modo que si solo hay una casilla legal no se realiza
// recursión, poniendo en ella directamente sin necesidad de saber su
// valoración.
// De igual modo podemos comprobar si la situación del tablero es la de
// apretura, para poner en cualquiera de las cuatro casillas, al ser
// simétricas.
// No debe ser llamada si el color especificado pasa.
        int numPos;             // Número de posiciones donde se puede poner.
        int contrario;          // Color contrario.
        int cont;               // Contador para recorrer las posiciones.
        int mejorVal;           // Mejor valoración de los movimientos evaluados.
        Posicion mejorPos;      // Mejor posición de las evaluadas.
        int aux;                // Variable auxiliar para guardar la valoración de
                                // la posición actual.
        CWaitCursor reloj;


        parar = false;
        numNodos++;                     // Un nodo más que se expande.
        contrario = colorContrario(color);
        gestor.Pedir();                 // "Pedimos" memoria.
        mejorVal = -MAXINT;
        tab.CasillasLegales (color, gestor.posicion(), numPos);
        if ((tab.NumeroBlancas() == 2) && (tab.NumeroNegras() == 2))
                // Situación inicial. Ponemos en cualquier posición, pues las
                // cuatro son simétricas.
                //mejorPos = gestor.posicion(random(4));
                mejorPos = gestor.posicion(0);
        else {
                if (numPos == 1)
                        // Solo hay un sitio donde poner. No perdemos el tiempo
                        // valorando la posición, pues no hay otra.
                        mejorPos = gestor.posicion(0);
                else {
                        // Hay varias posiciones legales, y no estamos en la primera
                        // jugada. Es el caso general, en el que sí hay recursión.
                        cont = 0;               // Empezamos a evaluar la primera casilla.
                        do {
                                // Bucle principal, que se repite mientras queden jugadas, o
                                // nos obligen a parar. Es un do en vez de un while, para que
                                // al menos se ejecute una vez, y tengamos alguna valoración
                                // que devolver si nos obligan a parar.
                                gestor.tablero() = tab;     // Copiamos el tablero original.
                                // Hacemos la siguiente jugada sobre el tablero.
                                gestor.tablero().HazJugada(gestor.posicion(cont), color);
                                // Hacemos la recursión. Cambiamos el signo al valor devuelto
                                // para evitarnos tener que distinguir el jugador siguiente
                                // para hacer el máximo o el mínimo. Eso nos obliga a que la
                                // función de evaluación estática dependa del turno.
                                // Debemos decidir que recursión hacer: minimax o alfabeta.
```

```
                                        if (podar)
                                                aux = -Alfabeta (gestor.tablero(), contrario,
                                                                this->nivelRecursion - 1,
                                                                -MAXINT, -mejorVal);
                                        else
                                                aux = -Minimax (gestor.tablero(), contrario,
                                                                 this->nivelRecursion - 1);
                                        // Miramos si la posición es mejor, y nos la quedamos si
                                        // procede.
                                        if (aux > mejorVal) {
                                                mejorVal = aux;
                                                mejorPos = gestor.posicion (cont);
                                                // No nos preocupa el valor de alfa o beta, pues estamos en
                                                // la raíz. No hay nodos superiores minimizadores, luego
                                                // beta siempre es "infinito". No hay nodos superiores
                                                // maximizadores, luego alfa solo depende del nodo actual
                                                // (raíz), y siempre será el mejor valor actual.
                                        }
                                        cont++;
                                } while ( (cont < numPos) && !parar);

                        }       // if que miraba si había o no solo una posición.

                }       // if que miraba si estabamos o no en la posición inicial.

                gestor.Liberar();
                return (mejorPos);

} // Pensar


int Pensador :: Alfabeta (Tablero &tab, int color, int nivelRec,  int alfa, int beta) {
// ...
                int numPos;     // Número de posiciones donde puede poner el color en
                                // el tablero.
                int cont;       // Contador, para recorrer las posiciones.
                int contrario;  // Color contrario a color.
                int mejorVal;   // Mejor valoración para el nodo actual.
                int aux;        // Variable auxiliar.
                int a, b;       // Valores alfa y beta que se propagarán a los hijos.
                bool poda;      // A cierto si se debe podar el resto de hijos que
                                // faltan por evaluar.

                numNodos++;
                if (FuncionTerminar (tab, color, nivelRec)) {
                        // Si hay que acabar la recursión, devolvemos la valoración de la
                        // posición por la función de evaluación actual.
                        numHojas++;
                        return(FuncionEvaluacion(tab, color));
                }
                else {
                        contrario = colorContrario(color);
                        if (!tab.Pasa(color)) {
                                // El color tiene posiciones donde poner. Hacemos el minimax.
                                gestor.Pedir();    // "Pedimos" memoria.
                                // Obtenemos las posiciones donde puede poner el jugador.
                                tab.CasillasLegales (color, gestor.posicion(), numPos);
                                mejorVal = -MAXINT;     // La peor valoración posible.
                                cont = 0;                       // Empezamos a recorrer desde la primera
                                                                // posición posible.
```

```
                poda = false;                   // No hay que podar los nodos hijos.
                a = alfa;
                b = beta;
                do  {
                        // Bucle principal, que se repite mientras queden jugadas,o
                        // nos obliguen a parar. Es un do en vez de un while, para
                        // que al menos se ejecute una vez, y tengamos alguna valo-
                        // ración que devolver si nos obligan a parar.
                        gestor.tablero() = tab;       // Copiamos el tablero original.
                        // Hacemos la siguiente jugada sobre el tablero.
                        gestor.tablero().HazJugada (gestor.posicion(cont), color);
                        // Hacemos la recursión. Cambiamos el signo al valor
                        // devuelto para evitarnos tener que distinguir el jugador
                        // siguiente para hacer el máximo o el mínimo. Además hay
                        // que intercambiar los valores alfa y beta, y cambiarlos
                        // de signo.
                        // Todo esto nos obliga a que la función de evaluación
                        // estática dependa del turno.
                        aux = -Alfabeta (gestor.tablero(), contrario, nivelRec - 1,
                                        -b, -a);
                        if (aux > mejorVal) {
                        // Si hemos encontrado un movimiento mejor, nos lo quedamos
                                mejorVal = aux;
                                // Debemos actualizar el valor alfa que propagamos
                                // a los hijos. El valor alfa es el máximo valor parcial
                                // de todos los nodos "maximizadores" superiores, por lo
                                // que propagaremos el máximo entre el que hemos recibido
                                // de arriba, y el que hemos conseguido en este nodo.
                                a = (a>mejorVal)?a:mejorVal;
                        } // if

                        // Comprobamos si se podemos podar.
                        if (beta <= mejorVal)
                                // Si el beta es menor, algún nodo "minimizador" superior
                                // tiene ya un valor parcial menor que el valor maximizador
                                // que estamos hayando, por lo que desechará nuestra
                                // evaluación, y podemos evitarnos seguir.
                                poda = true;

                        cont++;

                } while ( (cont < numPos) && !poda && !parar);
                gestor.Liberar();
                return (mejorVal);
        } // if que miraba si se pasaba o no.
        else
                // El jugador pasa. Llamamos recursivamente.
                return (-Alfabeta (tab, contrario, nivelRec - 1,
                                -beta, -alfa));

    } // if que miraba si era el caso base o recursivo.

} // Alfabeta
```

5.- AGRADECIMIENTOS

Queremos agradecer muy especialmente la ayuda prestada por Thomas Wolf. Él nos ha explicado con detalle, incluso por correo electrónico, casi todas las técnicas incrementales que hemos utilizado en nuestro programa que nos han llevado a conseguir la profundidad suficiente en la búsqueda.

También estamos obligados a dar las gracias a Olivier Arsac y sus compañeros, por su programa Darwersi que nos ha proporcionado buenas ideas para el interfaz del programa y para las funciones de evaluación. Sus ideas sobre las técnicas incrementales son también bastante buenas, aunque han llegado a nosotros un poco tarde para ponerlas en práctica.

Muchas gracias también a Andre Rassat y Matthieu Klein por su programa “Lothello” que nos ha dado alguna idea sobre la función de evaluación.

A todos ellos se debe que este programa haya podido llegar a ser así. ¡Gracias!