Revista Digital
Scientia Omnibus Portus
ISSN 2792-6885 Volumen 1 - Número 2 (Noviembre 2021)

Tres aplicaciones prácticas para entender la recursividad

Alfredo Moreno Vozmediano
Departamento de Informática, IES Celia Viñas (Almería, España)
alfredomoreno@iescelia.org

Recibido el 26/10/2021
Aceptado el 19/11/2021

Abstract

La recursividad es una de las técnicas de programación más potentes y a la vez más difíciles de manipular que existen. Es como tener un explosivo inestable entre las manos: algo tremendamente potente pero también peligroso (en términos de la estabilidad del programa).

Los estudiantes de programación suelen enfrentarse al problema de que la recursividad es un concepto sencillo de entender pero difícil de aplicar a problemas reales. En este artículo vamos a exponer tres ejemplos de soluciones recursivas de dificultad creciente, desde el trivial cálculo del factorial hasta un complejo algoritmo minimax con poda alfa-beta para que el ordenador juegue de manera inteligente al ajedrez, pasando por el clásico problema de las ocho reinas.

Estos tres problemas de naturaleza recursiva nos servirán para mostrar la manera en la que un programador o programadora debe proceder a la hora de razonar una solución recursiva y la elegancia inherente a estas soluciones.

Palabras clave: Recursividad, algorítmica, backtracking, minimax, teoría de juegos

1. ¿Qué es la recursividad?

Un programa informático no es más que la traducción a un lenguaje de programación de un algoritmo.

Cualquier algoritmo o subalgoritmo puede llamar (o invocar) a cualquier otro subalgoritmo, y este a otro, y así sucesivamente. En concreto, el subalgoritmo llamado puede ser el mismo que el algoritmo llamante. Dicho de otro modo, *un algoritmo o subalgoritmo se puede invocar a sí mismo*. Se dice entonces que el algoritmo *es recursivo* [1].

Un algoritmo puede ser recursivo directamente (cuando se invoca a sí mismo) o indirectamente (cuando invoca a otro y, después de un número indeterminado de invocaciones, se termina por invocar de nuevo al primero).

Planteamiento genérico de una solución recursiva

Un problema tiene una naturaleza recursiva cuando su solución se define en términos de sí mismo. Por eso, los problemas recursivos también son siempre iterativos. De hecho, cualquier problema resuelto mediante recursividad puede también resolverse con bucles, pero la afirmación contraria no siempre es cierta. Lo que ocurre es que la solución con bucles suele ser más larga y farragosa.

Para resolver un problema mediante recursividad debemos plantear siempre una condición (si ... entonces) dentro del algoritmo recursivo. La condición debe contemplar dos casos:

- El *caso general*, que realizará una llamada recursiva para hallar la solución.
- El *caso base*, en el que no es necesario hacer una llamada recursiva porque la solución es conocida.

El caso base es comparable a la condición de salida de un bucle: si no lo escribimos o está mal planteado, obtendremos una recursión infinita, cuyo resultado suele ser un fallo grave del programa (desbordamiento de pila, agotamiento de la memoria, etc [2]).

Ventajas e inconvenientes de las soluciones recursivas

Entre las ventajas de las soluciones recursivas podemos destacar las siguientes [3]:

- Hay problemas intrínsecamente recursivos cuya solución iterativa, aún siendo posible, resulta artificial y enrevesada.
- Aún en el caso de que no sea así, la recursividad es una herramienta de enorme potencia cuyas soluciones resultan más sencillas y elegantes que sus equivalentes iterativas. Esto, que puede parecer un capricho gratuito, es importante en ciertos problemas complejos.
- La recursividad es fundamental en algunos ámbitos: por ejemplo, los problemas relacionados con la inteligencia artificial y los sistemas expertos suelen tener soluciones recursivas por naturaleza.
- Hay programadores y programadoras a quienes les resulta más fácil pensar de manera recursiva que iterativa.

Por supuesto, la recursividad también presenta algunos inconvenientes [4]:

- Las soluciones recursivas tienden a ser menos eficientes que sus equivalentes iterativas, es decir, consumen en general más recursos de la máquina (más tiempo de CPU, más memoria, etc).
- La recursión infinita es un error de ejecución particularmente grave que puede comprometer la estabilidad y/o el rendimiento de todo el sistema.

Normas para diseñar correctamente un algoritmo recursivo

La recursividad mal aplicada provocará rápidamente un desbordamiento de pila. El programa dejará de funcionar y su consumo de recursos de la máquina crecerá rápidamente.

Para asegurarnos de que una solución recursiva está bien planteada debemos preguntarnos si cumple estas dos normas básicas:

- Existe al menos una condición de terminación (caso base) para el cual la solución no es recursiva.
- Cada invocación recursiva se realiza con un dato cada vez más próximo al caso base.

Estas dos normas por sí solas no garantizan que una solución recursiva sea correcta pero, a la inversa, sí podemos decir que una solución recursiva que no las cumpla será sin duda incorrecta.

2. Un ejemplo sencillo de recursividad: cálculo del factorial de un número natural

Un ejemplo clásico y sencillo de comprender de solución recursiva es el cálculo del factorial de un número natural.

El factorial de un número n (escrito n!) se calcula como:

$$n! = n \cdot (n - 1) \cdot (n - 2) \cdot ... \cdot 2 \cdot 1$$

Por ejemplo, el factorial de 6 es:

$$6! = 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$$

Este problema se presta a una solución iterativa, con un bucle que recorra los números desde 6 hasta 1 (o al revés) y un acumulador que vaya almacenando las multiplicaciones sucesivas. Pero también se puede plantear de manera recursiva mediante inducción, es decir, definiendo el problema en términos de sí mismo:

- *Caso general*: el factorial de un número n es n multiplicado por el factorial de n – 1. Esto es, para calcular un factorial (el de n) es necesario calcular otro (el de n – 1). Eso es lo que significa que problema se defina "en términos de sí mismo". Expresado matemáticamente:

$$n! = n \cdot (n - 1)!$$

- *Caso base*: el factorial de 1 es 1

Por ejemplo, el factorial de 6 se puede expresar de manera recursiva así:

$$6! = 6 \cdot 5!$$

Como vemos, en la definición de la solución de 6! aparece otra vez el factorial, solo que aplicado a un número menor (5!). A su vez, el factorial de 5 se puede calcular como:

$$5! = 5 \cdot 4!$$

Y así sucesivamente:

$$4! = 4 \cdot 3!$$

$$3! = 3 \cdot 2!$$

$$2! = 2 \cdot 1!$$

Finalmente, 1! es el *caso base* de la recursión, y su solución no necesita del cálculo de otro factorial, sino que podemos decir directamente que es 1. Siempre es necesario alcanzar un caso base en el que la solución se defina directamente para evitar una recursión infinita.

Transcribiendo esta idea a pseudocódigo tendríamos la versión recursiva de la función factorial:

```
Función factorial(n es entero): devuelve Entero
   Definir result como Entero
   Si n == 1 entonces
      resultado = 1                    // Caso base
   SiNo
      resultado = n * factorial(n-1)   // Caso general
   FinSi
   Devolver resultado
FinFunción
```

3. Un ejemplo más complejo de algoritmo recursivo: el problema de las ocho reinas

Una de las estrategias de búsqueda de soluciones a problemas complejos donde la recursivad muestra todo su poder es en el *backtracking* o búsqueda con retroceso [5]. Se trata de una estrategia recursiva de resolución de problemas cuyos resultados pueden llegar a ser espectaculares. Vamos a ver en qué consiste y la vamos a aplicar para resolver un problema clásico en programación: el de las ocho reinas.

El *backtracking c*onsiste, en pocas palabras, en ir incrementando una solución parcial con todos los posibles elementos que la amplíen hasta alcanzar la solución total. Las soluciones parciales deben entenderse como un conjunto de elementos que pueden ir calculándose en varios pasos, todos ellos similares entre sí, lo que hace que aparezca la recursividad en los programas desarrollados con este método.

Una codificación esquemática de esta idea puede ser la siguiente:

```
Función backtracking(solución es <tipo-de-datos>)
   Definir solución_encontrada como Booleano
   Si solucion es ya una solución general
     solución_encontrada = verdadero
   SiNo
     solución_encontrada = falso
   FinSi
   Repetir
     solución_parcial = solución parcial que amplíe solución;
     solución_encontrada = backtracking(solución combinada con solución_parcial)
   Hasta que solucion_encontrada = verdadero o no existan más soluciones parciales
   Devolver solución_encontrada;
FinFunción
```

Una variación habitual de este esquema consiste en encontrar *todas* las soluciones a un problema. En este caso hablamos de *algoritmos de fuerza bruta* y, aunque resultan altamente ineficientes, puede ser perfectamente viables si el espacio de soluciones del problema no es excesivo.

Una solución recursiva al problema de las ocho reinas

Existen muchos problemas clásicos en algorítmica, pero uno de los más célebres es el de las ocho reinas [6]. Consiste en colocar 8 reinas (o damas) en un tablero de ajedrez de modo que no se coman unas a otras.

Vamos a construir un algoritmo que resuelva este problema mediante *backtracking*, y que además sea escalable para poder generalizar la solución a la colocación de n-reinas en un tablero de tamaño n x n. Pero, antes de estudiar el algoritmo, sería aconsejable que el lector o lectora meditase un rato en cómo resolverlo sobre un tablero real. Verá que no es un problema en absoluto trivial.

Para nuestra solución, en lugar de representar el tablero con un array de 8x8 elementos (que es la primera idea que se nos podría ocurrir), usaremos un vector llamado *damas* y compuesto por 8 números enteros. El elemento i-ésimo de ese vector indicará en qué columna está la dama que hemos colocado en la fila i. Esta representación es pertinente porque debe haber exactamente una dama en cada fila (si no, se atacarían entre sí). Además, será más eficaz para nuestro propósito, como enseguida veremos.

Consideraremos como soluciones parciales el hecho de haber colocado i damas en las i primeras filas. La solución total consistirá en colocar 8 damas en las 8 primeras filas. Una solución parcial que amplíe a otra solución parcial será colocar i+1 damas en las i+1 primeras filas. Se ve aquí claramente cómo esta solución recursiva puede generalizarse para colocar n-reinas.

Comenzaremos por inicializar el vector *damas* a -1, así como i = 0. Para buscar la primera solución parcial, simplemente colocaremos la dama de la fila i en la columna 0. Entonces comprobaremos si la solución parcial es válida.

Si es así (y debe serlo, pues la primera dama nunca se comerá a sí misma), buscaremos la siguiente solución parcial, colocando la dama de la fila i+1 (es decir, la de la fila 1) en la columna 0. Comprobaremos si esta solución parcial es válida. Si resulta que no (como, de hecho, sucederá), probaremos con la siguiente columna, y luego con la siguiente, y así sucesivamente hasta hallar una solución parcial válida.

De ese modo proseguiremos con las 8 damas. Si alguna de ellas no es capaz de localizar una solución parcial válida, terminamos la recursión y probamos con otras combinaciones de columnas (*backtracking* o vuelta atrás).

Expresado en pseudocódigo, este algoritmo que acabamos de describir puede quedar así:

```
Algoritmo probar_ocho_reinas
    Definir damas como Vector[8] de Entero
    Definir solución como Booleano
    Escribir "Voy a colocar 8 reinas en un tablero de ajedrez. ¡Deséame suerte!"
    Para i desde 0 hasta 7 hacer        // Inicializamos las posiciones a -1
           damas[i] = -1
    FinPara
    solución = ocho_reinas(damas, 0);
    Si solución = verdadero entonces    // Mostramos la solución encontrada
       Para i desde 0 hasta 7 hacer
           Escribir "La reina nº ", i, " se coloca en:"
           Escribir " -fila ", i
           Escribir " -columna ", damas[i]
       FinPara
    SiNo
       Escribir "¡No he encontrado ninguna solución!"
FinAlgoritmo
```

```
// Esta función implementa el backtracking para encontrar las 8 reinas.
// El vector "damas" debe pasarse por referencia, no por valor

Función ocho_reinas(damas es Vector[8] de Entero [E/S], i es Entero):
devuelve Booleano
{
    Definir solución como Booleano
    Si i = 8 entonces    // Caso base
        solución = verdadero
    SiNo                 // Caso general
        solución = falso
        Repetir
                damas[i] = damas[i] + 1
                Si posición_valida(damas, i) == verdadero Entonces
                    solucion = ocho_reinas(damas, i+1)
                FinSi
        Hasta que solución = verdadero o damas[i] > 7
        Si solución == false Entonces
           damas[i] = -1    // No hemos encontrado solución para esta fila!
        FinSi
    FinSi
    Devolver solución
FinFunción
```

```
// Comprueba si una solución parcial es válida
// Sólo hay que comprobar la columna y las diagonales

Función posición_válida(damas es Vector[8] de Entero, i es Entero):
devuelve Booleano
    Definir válida como Booleano
    válida = verdadero
    Para j desde 0 hasta i-1 hacer
        // No es atacada en la columna
        válida = válida y (damas[i] != damas[j])
        // No es atacada en una diagonal
        válida = válida y ((abs(i-j)) != (abs(damas[i]-damas[j])))
    FinPara
    Devolver válida;
FinFunción
```

4. Una aplicación compleja de la recursividad: minimax con poda alfa-beta aplicado a un juego de ajedrez

Un ejemplo de aplicación más compleja de la recursividad, y donde encontrar una solución iterativa sería francamente complicado, es el ajedrez. En lo que sigue, vamos a discutir un caso particular dentro de este fascinante campo: el algoritmo minimax con poda alfa-beta aplicado a un juego de ajedrez, aunque podríamos aplicarlo a muchos otros juegos de dos jugadores en los que necesitemos que el programa informático se comporte de manera inteligente (lo cual no es exactamente lo mismo que afirmar que el programa *sea* inteligente, pero esta es otra discusión).

La evaluación estática

Para que un ordenador pueda jugar al ajedrez razonablemente bien, debe empezar por ser capaz de distinguir las situaciones más beneficiosas (para él) de las que no lo son tanto. Por ejemplo, debe saber que tener un peón en el centro del tablero es mucho mejor que tenerlo en un lateral, o que la reina es mucho más valiosa que un caballo.

Para lograrlo existen varios métodos, pero uno sencillo y eficaz es la *función de evaluación estática* o *función posicional*. Se trata de una función matemática que, partiendo de las piezas que hay en el tablero y su ubicación, devuelve un valor numérico que determina la calidad de la situación para un jugador dado.

Existen muchas funciones de evaluación disponibles en la bibliografía especializada [7]. Las mejores son las que combinan el resultado de la evaluación con otras consideraciones (como bases de datos de partidas históricas, el momento actual del juego -apertura, juego medio, final-, la estrategia global del jugador, etc) pero, para nuestro propósito, una función estática simple es más que suficiente y logrará que nuestro algoritmo juegue competentemente.

Las funciones de evaluación estática más simples sugieren reglas de este estilo:

- Por cada peón propio, sumar 100 puntos. Por cada peón del contrincante, restar 100 puntos.
- Si el peón está en el centro del tablero, añadir o restar 20 puntos más por cada uno.
- Sumar o restar 2 puntos por cada casilla que haya avanzado el peón desde su punto de partida.
- Etc.

Y así sucesivamente para cada uno de los tipos de pieza. El resultado de la función debe ser un número entero positivo para posiciones favorables al jugador (cuanto más grande, mejor para ese jugador) y un número entero negativo para posiciones desfavorables (cuando más pequeño, peor para ese jugador).

Así, dada una posición cualquiera del tablero, el ordenador puede efectuar muchos movimientos diferentes. Un enfoque inteligente consiste en buscar todos los movimientos posibles y, para cada uno de ellos, calcular en qué situación queda el tablero según la función de evaluación estática. Al final, el programa escogerá el movimiento que le conduce a una situación en la que el valor devuelto por la función de evaluación es máximo.

Técnicas heurísticas

Se denominan *técnicas heurísticas* aquellas que no nos aseguran encontrar una solución perfecta a un problema (en nuestro caso, una solución perfecta sería encontrar un movimiento que nos condujera directa e inevitablemente a ganar la partida de ajedrez), pero sí hallar una solución de la que se puede esperar que esté entre las mejores soluciones posibles.

Con la función de evaluación estática, y siempre que dicha función sea lo bastante buena, conseguiremos que el ordenador "piense" su siguiente movimiento siguiendo criterios objetivos, pero cometerá continuamente torpezas como, por ejemplo, sacrificar una reina para comer un peón. Esto se debe a que, a diferencia de los jugadores humanos, el programa informático no verá más allá del siguiente movimiento, mientras que un humano intentará predecir la reacción de su adversario en los siguientes dos, tres, cuatro o más movimientos.

Por supuesto, ningún ser humano puede calcular todos los posibles movimientos de las siguientes dos, tres o cuatro jugadas, y menos aún cómo acabará la partida. Ello se debe a la enorme cantidad de posibilidades que se

abren desde cada posición del tablero. Son tantas que ni siquiera un ordenador potente puede evaluarlas todas más allá de los siguientes tres o cuatro movimientos.

Para hacernos una idea de la enorme amplitud del dominio del problema, se calcula que el número de posibles combinaciones de piezas después de los diez primeros movimientos es del orden de 10^{27} [8]. Un ordenador capaz de evaluar, por ejemplo, un millón de posiciones por segundo, necesitaría más de 7 billones de años (500 veces la edad de nuestro universo) para generar todas las jugadas posibles y decidir cuál es la mejor.

Ante la imposibilidad de una evaluación exhaustiva de todas las jugadas posibles a partir de un estado concreto, tenemos que conformarnos con un examen parcial. Una estrategia para lograr una buena heurística es la técnica conocida como *minimax*, que se puede usar en otros muchos juegos de dos jugadores para hacer que el ordenador tome decisiones inteligentes.

Minimax

Supongamos que el ordenador maneja las piezas negras. Lógicamente, su siguiente movimiento debería ser aquél que haga máxima la función de evaluación para las piezas negras. Pero debemos tener en cuenta que el contrincante (piezas blancas) responderá con la jugada que haga máxima la función de evaluación para las blancas, es decir, que la convierta en mínima para las negras.

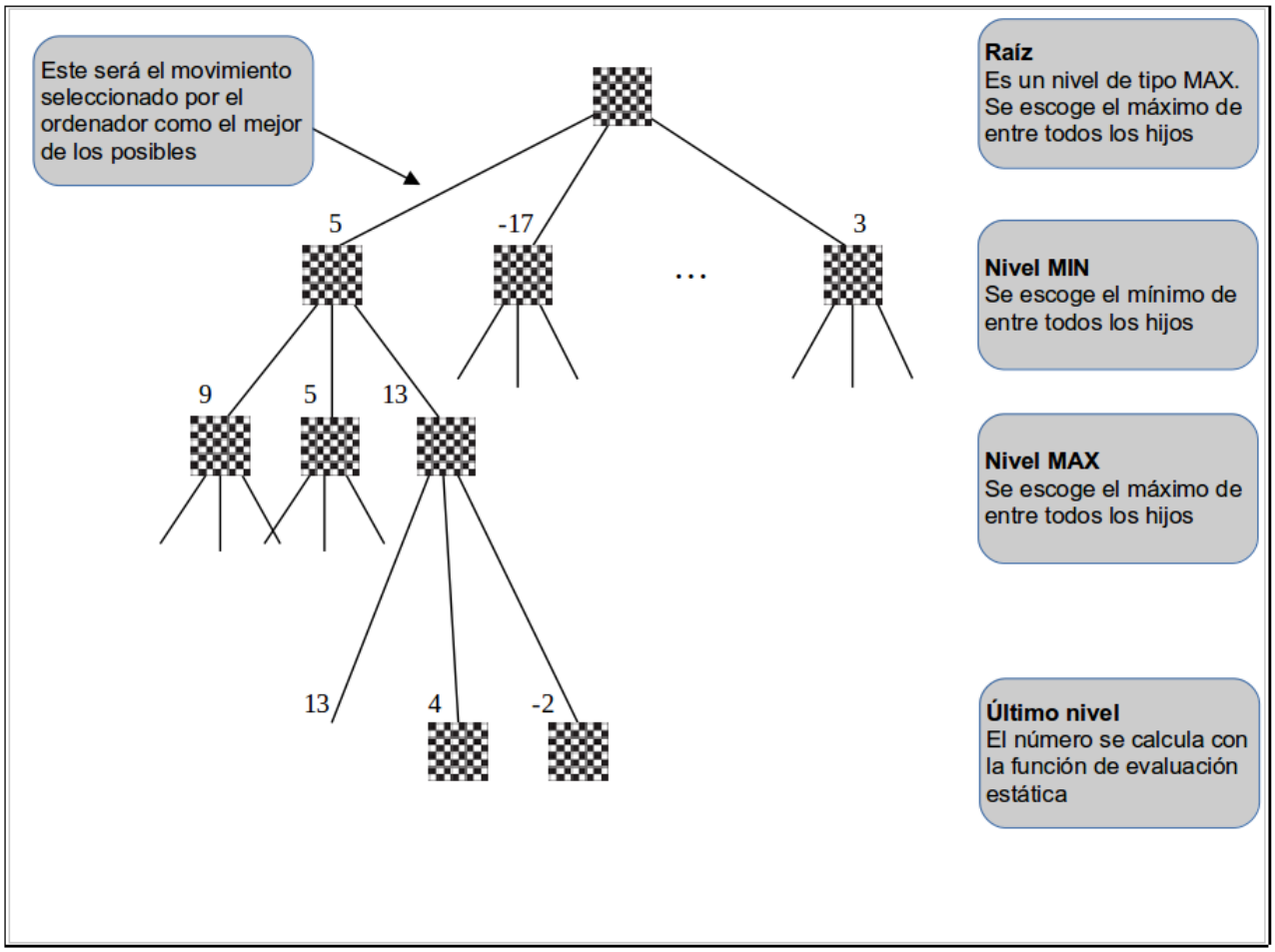
Si exploramos la función de evaluación estática no para el siguiente movimiento, sino para los *dos siguientes*, el programa ya no cometerá torpezas como sacrificar una reina para comer un peón. Aunque eso haga máxima la función de evaluación en el siguiente movimiento, la hará mínima dentro de dos, porque podemos prever que el contrario preferirá comerse nuestra reina y, por lo tanto, no elegiremos ese curso de acción.

Naturalmente, pudiera ocurrir que el movimiento elegido de esta forma sea equivocado, en el sentido de que no conduzca a una jugada ganadora. Al fin y al cabo, sólo estamos comprobando la respuesta del contrario a cada posible movimiento nuestro, es decir, sólo estamos prediciendo dos movimientos en el futuro. Quizá mirando cinco o seis movimientos más adelante nos diéramos cuenta de que no es rentable, pero eso es imposible, como hemos visto, por la enorme amplitud del espacio de estados del ajedrez. Por eso esta estrategia no nos asegura la victoria, pero sí incrementa nuestras posibilidades de alcanzarla.

La técnica minimax consiste en descender en el árbol de movimientos tanto como la potencia de cálculo de la máquina nos permita en un tiempo razonable, tratando en cada nivel que el jugador actual (que maneja el ordenador) haga máxima su ventaja al tiempo hace mínima la ventaja del jugador contrario [9]. Por esa razón esta técnica se denomina *minimax*.

En la siguiente figura lo expresamos gráficamente. Supongamos que el juego se encuentra en el estado representado en la raíz del árbol, y que le toca mover al ordenador. Cada posible movimiento conduce al tablero a un estado diferente en el siguiente nivel del árbol. A su vez, cada uno de estos tableros tiene una colección de posibles movimientos que generan otros estados del tablero.

Los valores asignados a los estados más bajos (hojas del árbol) se obtienen aplicando la función de evaluación estática. El resto de valores se obtienen mediante la regla minimax. El jugador negro elegirá el primer movimiento (el de la rama izquierda), porque le asegura un valor mínimo de 5. Es decir, es el máximo de los valores mínimos del siguiente nivel.



Aplicación del minimax recursivo a un juego del ajedrez

Para aplicar el algoritmo minimax a un juego de ajedrez se pueden seguir muchas estrategias pero, en general, necesitaremos implementar dos funciones como estas:

- *valorar_posición_contrario()*: dada una posición del tablero, esta función realizará la evaluación estática si ya se ha alcanzado la profundidad máxima de búsqueda y devolverá esa valoración. Si no se ha alcanzado esa profundidad, se generarán todos los posibles movimientos del jugador contrario y se llamará a la función *valorar_posición_propia()*, quedándonos con el valor mínimo de todos los que nos devuelva esta función.
- *valorar_posición_propia()*: dada una posición del tablero, realizará la evaluación estática si ya se ha alcanzado la profundidad máxima de búsqueda y devolverá esa valoración. Si no se ha alcanzado esa profundidad, se generarán todos los posibles movimientos del jugador que tiene el turno y se llamará a la función *valorar_posición_contrario()*, quedándonos con el valor máximo de todos los que nos devuelva esta función.

En definitiva, la función *valorar_posición_contrario()* se ejecutará en los niveles del árbol correspondientes al jugador contrario (MIN) y *valorar_posición_propia()* se ejecutará en los niveles del árbol correspondientes al jugador actual (MAX)

Obsérvese que la primera función llama a la segunda y la segunda a la primera, por lo que estamos usando una *recursión indirecta*. Sería muy complicado implementar esta solución mediante iteraciones. El *caso base* es aquel en el que se alcanza la profundidad máxima preestablecida: entonces se invoca la función de evaluación estática y la recursión termina.

Una mejora: minimax con poda alfa-beta

La técnica minimax, como hemos dicho, no tiene por qué proporcionarnos el mejor movimiento posible. De hecho, puede provocar un movimiento manifiestamente malo y ante el que cualquier jugador avanzado de ajedrez sonreiría con suficiencia. Aunque, si profundiza lo suficiente en el árbol de movimientos, la mayoría de las veces minimax proporcionará un movimiento razonablemente bueno.

El movimiento será tanto más bueno cuanto más podamos profundizar en el árbol. De hecho, se considera que un buen jugador de ajedrez suele prever, aproximadamente, una media de 8 jugadas. Pero, como hemos visto, este árbol se ramifica demasiado y se hace muy pronto incalculable, incluso para los ordenadores más potentes. A este tipo de problemas se les denomina *no computables*, ya que no pueden ser procesados y resueltos en un tiempo razonable.

Pero hay una forma de lograr profundizar más en el árbol, llegando hasta seis, siete o más jugadas en el futuro: utilizando una variación de la técnica minimax denominada *minimax con poda alfa-beta* [10].

El minimax con poda se basa en la misma estrategia que utiliza un jugador humano de ajedrez: no es necesario mirar *todos* los estados porque hay algunos que, ya desde el principio y de forma evidente, son malos para el jugador que mueve. Por lo tanto, esas ramas del árbol se desechan porque, por más que descendamos, todos los estados resultarán muy negativos.

Para aplicar este principio a nuestro algoritmo debemos ejecutar la función de evaluación estática en algún nivel intermedio, antes de llegar a las hojas del árbol. Por ejemplo, si estamos explorando 5 jugadas, podemos aplicar la función de evaluación en el nivel 3. En esa jugada, le toca mover al jugador actual (es un nivel de tipo MAX en el algoritmo minimax). Evaluaremos todos los estados del nivel 3 como si fueran los últimos del árbol, y llamaremos al mejor de ellos ALFA.

A partir de entonces sólo seguiremos explorando las ramas del árbol cuyo valor estático en el nivel 3 sea igual ALFA (o, al menos, muy próximo a ALFA). Esto eliminará la gran mayoría de ramas del árbol, por lo que nos será fácil descender hasta el nivel 7 u 8. Si hacemos una segunda poda, podemos explorar niveles muy profundos en un tiempo razonable.

También podemos elegir hacer la poda en un nivel correspondiente al jugador contrario (por ejemplo, el nivel 4, que es de tipo MIN). En ese caso, aplicaremos la función de evaluación estática a todos los estados y elegiremos la menor de todas ellas, llamándola BETA. Podaremos todas las ramas cuyo valor estático sea mayor que ese valor BETA, profundizando en las demás.

Evidentemente, este método tampoco es infalible pues, aunque permite profundizar mucho más en el árbol, es muy posible que en alguna de las podas desechemos una rama que, aunque en ese momento proporciona una valoración pobre, en el futuro nos hubiera conducido a la victoria. A pesar de este riesgo, inherente a todas las técnicas heurísticas (que, recordemos, no pretenden encontrar soluciones infalibles, ya que se usan en problemas para los que es imposible encontrar dichas soluciones), el minimax con poda alfa-beta suele proporcionar mejores resultados que el minimax simple.

5. Conclusiones

La recursividad es una técnica de programación conceptualmente muy simple y técnicamente muy poderosa que permite plantear soluciones elegantes a problemas intrínsecamente recursivos.

Aunque todos los problemas recursivos son también iterativos, las soluciones iterativas pueden llegar a ser mucho más complejas que sus equivalentes recursivas. Sin embargo, la recursión debe plantearse de manera adecuada para no comprometer la estabilidad del programa y del sistema donde se ejecuta, definiendo muy bien su caso base y su caso general y asegurándose que, en cada llamada recursiva, el programa se aproxima más al caso base.

La recursividad se puede aplicar con éxito a problemas muy simples, como el cálculo del factorial de un número natural, o a problemas muy complejos, como el diseño de un algoritmo inteligente para jugar al ajedrez, obteniendo soluciones elegantes y razonablemente fáciles de comprender incluso para programadores principiantes como las que hemos presentado en este texto.

Referencias

[1] Barron, David William, "Recursive techniques in programming", *Macdonald Computer Monographs (1 ed.)*, Londres, 1968, viii+64 pp.

[2] Kanetkar, Yashavant, "ANSI C Programming", BPB Publications, Nueva Delhi, 2019.

[3] Di Mare, Adolfo, "Tres formas diferentes de explicar la recursividad", *Revista Ingeniería*, Facultad de Ingeniería de la UCR, 1996, vol. 6, num. 2, pp. 31-44.

[4] Quintana M.E. et al., "Estrategias de enseñanza de funciones recursivas en ciencias de la computación", *Boletín Redipe*, 2016, pp. 83-91. [Online] https://revista.redipe.org/index.php/1/article/view/128

[5] Knuth, Donald E., "Introduction to Backtracking", *The Art of Computer Programming*, Vol. 4, Fasc. 5, Addison Wesley, 2020.

[6] Petkoviç, M., "Mathematics and Chess", Dover Publications, 2003.

[7] Drakos, N. y Moore, R. [Online], "Desarrollo de la Maestría Ajedrecística Computacional utilizando recursos restringidos", *Departamento de Ciencias de la Computación, Universidad de Chile*, Santiago, 1999. [Online] https://users.dcc.uchile.cl/~jegger/memoria/node51.html, Universidad de Chile

[8] Atashpendar, A., Schilling, T. y Voigtmann, Th., "Sequencing Chess", *Universidad de Cornell*, Ithaca, Nueva York, 2016. [Online] https://arxiv.org/abs/1609.04648

[9] Kjeldsen, T. H., "John Von Neumann's Conception of the Minimax Theorem: A Journey Through Different Mathematical Context", *Universidad de California en Santa Barbara*, 2001. [Online] https://web.math.ucsb.edu/~crandall/math201b/vnminimax.pdf

[10] González Labarta, R. A. et al., "Motor de juego de ajedrez Alessandro 1.0", *Universidad de Ciencias Informáticas*, La Habana, Cuba, 2014. [Online] https://www.researchgate.net/publication/260421267_Motor_de_Juego_de_Ajedrez_Alessandro_v10_Chess_Engine_Alessandro_v10

Biografía

Alfredo Moreno Vozmediano es Ingeniero Técnico en Informática de Sistemas por la Universidad de Castilla-La Mancha e Ingeniero Superior en Informática por la Universidad de Málaga. Trabajó durante dos años como programador en la consultora Atos ODS y en la desarrolladora de videojuegos Pyro Studios. Desde el año 2000, es funcionario de carrera en el Cuerpo de Profesores de Enseñanza Secundaria de la Junta de Andalucía. Ha publicado diversos libros de contenido técnico, como "Aprender a programar en C: de 0 a 99 en un solo libro" o "Java para novatos", además varios relatos y novelas de ciencia-ficción como "Arcadia" o la saga "Kepler-22b". Actualmente ejerce como profesor de informática en el IES Celia Viñas de Almería y compagina su actividad docente con la escritura de textos de ficción y no ficción.