5-2-2021

PRÁCTICA 2

Búsqueda con adversario

Ángel Ortega Alfaro & Verónica Giraldo Garrido
ANGEL.ORTEGA@ALU.UCLM.ES & VERONICA.GIRALDO1@ALU.UCLM.ES

ÍNDEX

Introducción

En esta práctica nuestro objetivo es diseñar el problema de ajedrez usando agentes que jueguen (piezas del tablero) de forma competente al ajedrez, utilizando las estrategias de búsqueda con adversario para planificar la secuencia de acciones que posteriormente será ejecutada por el agente para alcanzar el objetivo. Dicho objetivo consiste en, dado un tablero con sus piezas generado de manera aleatoria, simular una partida de ajedrez entre dos jugadores diferentes. Para ello, vamos a utilizar el algoritmo MiniMax y la poda Alfa-Beta.

El entorno de nuestro problema consistirá en un tablero de ajedrez, generalmente con un tamaño de 8x8 (las dimensiones no varían), sobre el cual las piezas se generarán de manera aleatoria, y dependiendo de la semilla que especifiquemos como parámetro a la hora de ejecutar el código y crear el tablero.

En la representación, se han numerado las filas y columnas de 0 a 7 (o de 0 a n-1 para cualquier otro caso). Las piezas negras se llaman con letras minúsculas y se mueven de abajo a arriba, y las blancas se llaman con mayúsculas y se mueven de arriba a abajo. En los diferentes tipos de piezas encontramos al peón (P), rey (K), reina (Q), torre (R), alfil (B) y caballo (N), nombradas mediante la notación en inglés.

```
   00 01 02 03 04 05 06 07
-------------------------
00|  |  |  |  |  |*Q|  |  |
-------------------------
01|  |  |  |  |  | p|  | p|
-------------------------
02|  | p| P| r|  |  | p|  |
-------------------------
03|  |  |  |  |  |  | q|  |
-------------------------
04|  |  |  |  |  |  |  |  |
-------------------------
05|  |  |  | p| P|  |  | P|
-------------------------
06|  |  | k|  |  | P| P|  |
-------------------------
07|  |  |  | R|  | K|  |  |
-------------------------
```

Imagen 1: Instancia del problema y representación

Podremos jugar tanto con las piezas blancas como las negras, empezando siempre a jugar MAX.

Como es imposible abordar el total de la profundidad de la búsqueda, daremos como parámetro de entrada el nivel máximo de profundidad de búsqueda (3 y 4 en este caso). Una vez alcanzada la profundidad máxima permitida, si no se ha alcanzado un estado terminal, deberemos evaluar la posición obtenida mediante una función de evaluación

o heurística, basada en valorar el número y tipo de piezas que hay presentes todavía en el tablero, tanto del agente al que le toca jugar como de su adversario.

Descripción de la implementación

State, Position, Action y ParActionValue

- **State.** State contiene como miembros la matriz bidimensional que representa el tablero, el tipo de pieza y el color que representan a nuestro agente. Implementa el método *isFinal()*, que devuelve true si el estado actual es final (nuestro agente se ha comido al rey enemigo); el método *copy()*, que devuelve una copia idéntica del estado actual sin modificarlo; el método de heurística, que devuelven la utilidad correspondiente dependiendo del color del agente que ha ganado la partida; el método *applyAction(Action action)*, que devuelve el estado obtenido al aplicar la acción pasada sobre el actual, y finalmente está el método *getPossibleActions (int white)*, que devuelve todos los posibles movimientos de una pieza en concreto de acuerdo a su color.
    - El estado actual no se modifica puesto que se aplica la acción sobre una copia haciendo uso del método *copy()*.
    - El método usado para obtener el valor de heurística de ese estado en concreto.
- **Position.** La clase Position simplemente representa una celda de nuestro tablero mediante la fila (row) y columna (col) correspondiente.
- **Action.** La clase Action representa la posición inicial y la posición final del agente tras ejecutarse la acción. Además, se implementa el método *getCost()*, que nos devuelve el coste de la acción aplicada según la expresión anterior. También nos encontramos con un *toString()*, que imprime la cadena de posiciones iniciales y finales de las piezas para comprobar el correcto funcionamiento de cada uno de los algoritmos.
- **ParActionValue.** En esta clase guardamos la acción elegida de la pieza (cada vez que se mueve) y el valor de la heurística asociada a dicha acción.

Piece y subclases

La clase **Piece** define las variables básicas de *color* y *tipo* para cada pieza, e implementa tantos métodos como movimientos pueden realizar las distintas piezas que representan los elementos del tablero de ajedrez. Cada subclase representa una pieza. A excepción del peón (tiene sus movimientos definidos en su propia clase), todas las demás piezas están formadas por un *ArrayList*, al cual se le añaden todos sus posibles movimientos dependiendo de la posición en la que se encuentre la pieza.

- En esta ocasión, en Piece se ha añadido una nueva variable para que tomase en cuenta el color de la pieza en cuestión, de tal forma que así pudiera comerse a las piezas del otro color.
- Para Pawn, hemos duplicado y adaptado el código para que las piezas que vayan a moverse sean tanto blancas como negras, y no únicamente un color fijado.

BusquedaAdversario y algoritmos

La clase **BusquedaAdversario** se encarga de agrupar la llamada a cada juego. Se pasa como argumento el nombre del algoritmo deseado (*MiniMax* o *Alfabeta*), el número de la pieza correspondiente con la que queramos jugar, la profundidad máxima establecida, el color del agente que moverá la pieza, una semilla fija de valor 1, una semilla variante que alternará valores entre 1 y 10 para generar distintos escenarios, el tipo de juego si jugamos contra la máquina o juega la máquina sola y el número de jugadas.

Dentro de esta clase, se llama a un switch con el que definiremos el color del agente en cuestión, y otro switch más en el que elegiremos finalmente qué juego se ejecutará.

En cuanto a las clases que representan los algoritmos, tenemos una clase que resuelve el tablero mediante el algoritmo por **MiniMax**, y otra clase que lo resuelve mediante el algoritmo por **poda con Alfa-Beta**. Serían dos clases en total, que corresponden con el primer parámetro que le pasamos a **BusquedaAdversario**.

- **Interfaz algoritmo:** esta interfaz se usa para que los algoritmos MiniMax y Poda tengan el método doSearch().
- **MiniMax** tiene declaradas las variables profundidad, color, tiempo y nodos. A su vez, también contiene el método **doSearch()** que inicia la búsqueda por Max, y va llamando a Max o Min según corresponda el turno. Además, en esta clase se encuentran también los métodos **Max_Valor(State estado, int prof, int color)** y **Min_Valor(State estado, int prof, int color)**, los cuales son idénticos salvo que en el primero nos quedamos con el valor máximo y en el segundo, con el valor mínimo.
    - El funcionamiento de la clase **MiniMax** consiste en generar los hijos del estado que nos han pasado por parámetro, comprobar si la profundidad es o no 0 para devolver el **ParActionValue**, y si la profundidad no es 0, se llama al método **Max_Valor** o **Min_Valor** según toque para comprobar el valor de la heurística en ese caso.
- **PodaAlfaBeta** es prácticamente una clase igual a la anterior. La única diferencia reside en que los métodos **Max_Valor** y **Min_Valor** de esta clase comprueban el valor de alfa o beta según corresponda.

Juegos contra la máquina

Para jugar al ajedrez que tenemos en el código, tenemos varias opciones:

- **Dummy**: Dummy es un modo de juego en el que sólo movemos nosotros, manteniéndose el adversario quieto ante nuestros movimientos. Este modo de juego se caracteriza por tener un método busca() en el que tenemos un bucle for con tantas iteraciones como número de jugadas que podemos hacer. Dentro del bucle, obtenemos el estado actual del tablero, sacando las posibles acciones o movimientos que podemos realizar. Mostramos estos movimientos por pantalla e indicamos cual de ellos queremos realizar. Aplicando esta acción, volvemos a obtener un estado nuevo en el que se nos vuelven a mostrar las posibles acciones en la siguiente iteración.
- **Todo**: Todo es un modo de juego en el que sólo juega la máquina contra sí misma. Al igual que Dummy, cuenta con un bucle for condicionado por el número de jugadas que se van a realizar. Dentro del for, se cuenta con una estructura if-

else que dependerá de si estamos buscando por MiniMax o por la poda. En lo que respecta a los movimientos en sí, se realiza la búsqueda y con la acción que se ha decido que es la mejor, se genera un estado el cual será del que se parta en el siguiente movimiento y desde donde deberá mover pieza el color contrario.

- **BlackWhite**: BlackWhite es un modo de juego en el que nosotros jugamos contra la máquina. Esta clase coge elementos de las dos explicadas anteriormente. Puesto que jugamos contra la máquina, debemos de indicar qué movimientos realizar, tal y como se explica de forma análoga en el apartado de Dummy. Una vez tengamos el estado elegido, se lo pasaremos a la máquina y ésta realizará la acción que mejor le venga acorde a la búsqueda MiniMax o a la poda Alfa-beta.

Función metaheurística

Contamos con una función heurística que se encarga de analizar el tablero de un estado dado y valorar el estado actual.

Esta función se encarga de recorrer el tablero, y según el tipo de pieza que encuentre, suma un valor u otro. Si encuentra un peón, suma o resta un 1; si encuentra una torre, un alfil o un caballo, suma o resta un 2; si encuentra una reina, suma o resta un 3 y por último si encuentra un rey suma o resta un 4. Una vez haya recorrido todo el tablero, resta el valor obtenido de las piezas negras y el valor obtenido de las piezas blancas si estamos minimizando negras, y al contrario si estamos minimizando blancas, devolviendo ese variable entera que se corresponde al valor de la heurística a ese estado.

En un primer momento, se consideró que el valor de cada pieza fuese siempre 1, es decir, que simplemente se contasen el número de piezas, sin embargo, rápidamente nos dimos cuenta que esta heurística no era lo suficientemente buena, ya que el número de piezas no es significativo a la hora de decidir si un estado es mejor o peor, por lo que decidimos darle un valor distinto a cada pieza.

Imagen 1: Ejemplo de heurísticas

Como podemos ver en la ilustración, tenemos un estado en el que tenemos el mismo número de piezas tanto blancas como negras. Hay una situación de peligro tanto para la torre blanca como para la reina negra, en la que según jueguen una u otra, se pueden eliminar de forma mutua. Si la función heurística simplemente contase el número de fichas, no tendríamos un valor de heurística muy acertado; sin embargo, al tener cada pieza un valor distinto, si la torre blanca se come a la reina negra, tendríamos un valor de heurística más distinto y que beneficiaría mucho más a las blancas.

Comparación del rendimiento

A continuación, tras estudiar los algoritmos pertinentes en esta práctica, hemos sacado una comparativa del algoritmo MiniMax en base a dos profundidades distintas, y hemos hecho lo mismo de acuerdo al algoritmo de poda. En general, comparamos resultados de los algoritmos por separado, ya que sus valores se encuentran en rangos y escalas muy distintos entre sí.

- Todos los datos han sido sacados a partir del modo de juego "Todo", en el cual la máquina juega consigo misma la partida. La búsqueda a fin de cuentas es igual para cualquier modo de juego, y cuando mueve el jugador humano no cuenta. El valor obtenido en cada escenario es una media de 10 ejecuciones sobre el mismo escenario, pero con una semilla distinta, para que los movimientos fuesen distintos y no se generase el mismo árbol de búsqueda.

Las gráficas generadas son las siguientes:

Escenario	Profundidad 3	Profundidad 2
Escenario 1	95217,3	2086,3
Escenario 2	80520,2	1952
Escenario 3	80630,3	2090,1

Imagen 2: Datos obtenidos para el algoritmo MiniMax

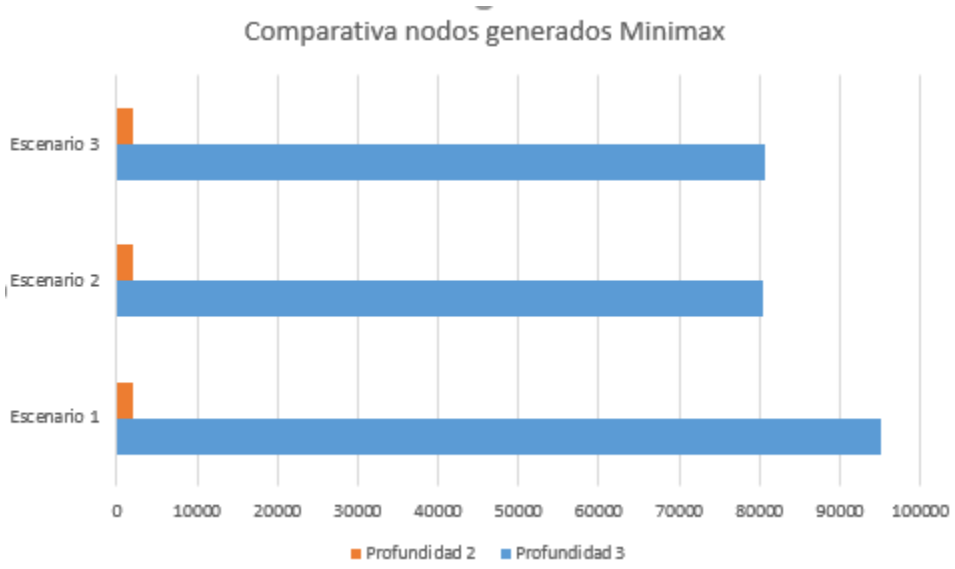

Imagen 3: Comparativa de nodos generados para el algoritmo MiniMax

Como podemos observar en esta primera gráfica, la profundidad es un factor de suma importancia a la hora de generar nodos: cuanto mayor profundidad exista, más nodos serán generados y más se expandirá el algoritmo en sí. Esto es debido a la cantidad de piezas disponibles sobre el tablero y la variedad de movimientos que podemos realizar con ellas.

Escenario	Profundidad 3	Profundidad 2
Escenario 1	45	30
Escenario 2	45	30
Escenario 3	45	30

Imagen 4: Datos obtenidos para el algoritmo Alfabeta (poda)

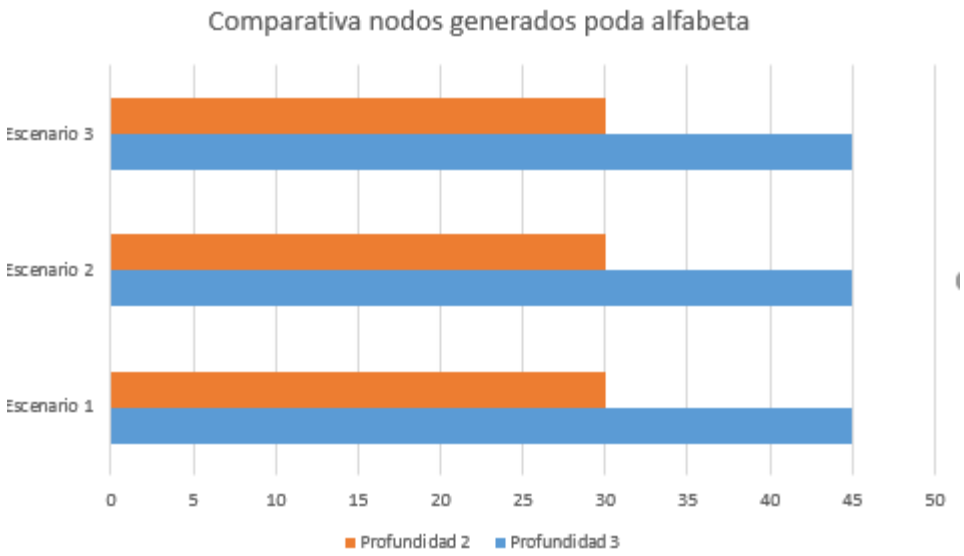

Imagen 5: Comparativa de nodos generados para el algoritmo Alfabeta (poda)

En esta segunda gráfica, la explicación de los resultados es análoga a la anterior. Según la profundidad aumente, el número de nodos generados aumentará también. Por otro lado, estos resultados son mucho menores en valor que los mostrados en el algoritmo MiniMax, porque en la poda el árbol de nodos que se genera es más pequeño. Esto es debido a que la función heurística no es la mejor que podemos usar, sin embargo, es eficiente para generar un resultado final como el de minimax.

Conclusiones

Tras el estudio de estos algoritmos y modos de juego del ajedrez, podemos concretar las siguientes conclusiones:

- El algoritmo de poda Alfabeta es mucho más eficiente que el algoritmo MiniMax, ya que al podar las ramificaciones del árbol con peor heurística obtenemos unos resultados mejores en menos tiempo. Es decir, el algoritmo no necesita recorrer todos los nodos generados para dar con una solución.
- A mayor profundidad pasada por parámetro, más cantidad de nodos generados encontraremos. En este caso, también el número de jugadas que queramos realizar es relevante, pues no obtendremos el mismo resultado con un único movimiento, por ejemplo, que con cinco o diez.