

# Simulación del juego QUARTO! y algoritmo Minimax

Diego Hernández Jiménez

curso 2020-2021

## Resumen

El objetivo de este trabajo es la simulación de una variante del juego QUARTO! y de un agente que, como mínimo, sea capaz de jugar empleando una estrategia mejor que el azar. Se propone el algoritmo de búsqueda minimax, más concretamente el algoritmo que incluye la técnica de poda alfa-beta. La implementación del juego se realiza en R, mientras que se utiliza C++, mediante el paquete `Rcpp`, para programar la estrategia minimax. Estas funciones se encuentran en un documento `.cpp` aparte, pero se describen cuando es necesario. Para comprobar la eficacia de la estrategia se simulan 100 partidas.

## Descripción y representación del juego

La información básica del juego ha sido obtenida de la web [ultraboardgames.com](http://ultraboardgames.com).

### Tablero y fichas

- Se dispone de un tablero de 4 x 4
- Se tienen 16 piezas que varían en cuatro atributos: las piezas pueden ser claras u oscuras, con base redonda o cuadrada, altas o bajas y macizas o con una perforación.

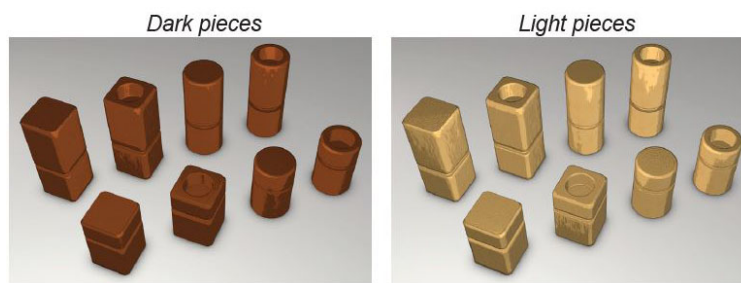


Figure 1: Piezas del juego. Fuente: [ultraboardgames.com](http://ultraboardgames.com)

Puede observarse que todas las combinaciones distintas de atributos están representadas. El conjunto de las piezas no es más que el producto cartesiano de los cuatro conjuntos de atributos de dos posibles propiedades cada uno. En R, es posible generar el producto cartesiano mediante la función `expand.grid`, o de manera más eficiente, de la siguiente forma:

```
1 create.pieces <- function(){
2   return(cbind(rep(1:2,each=8,times=1),
3               rep(3:4,each=4,times=2),
4               rep(5:6,each=2,times=4),
5               rep(7:8,each=1,times=8)))
6 }
7
8 create.pieces()
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    3    5    7
## [2,]    1    3    5    8
```

```
## [3,] 1 3 6 7
## [4,] 1 3 6 8
## [5,] 1 4 5 7
## [6,] 1 4 5 8
## [7,] 1 4 6 7
## [8,] 1 4 6 8
## [9,] 2 3 5 7
## [10,] 2 3 5 8
## [11,] 2 3 6 7
## [12,] 2 3 6 8
## [13,] 2 4 5 7
## [14,] 2 4 5 8
## [15,] 2 4 6 7
## [16,] 2 4 6 8
```

Cada fila es una pieza con una combinación diferente de propiedades. Se ha decidido identificar las propiedades con números porque todas las fichas son equivalentes, los atributos que se posean no son relevantes. Además, la identificación mediante números es ventajosa en por varios motivos. Por un lado, permite hacer operaciones matemáticas como sumar valores. Se verá más adelante su utilidad. Además, el acceso a elementos de una matriz numérica es más rápido que el acceso a elementos de un dataframe.

```
mat.num <- create.pieces()
df.num <- expand.grid(1:2,3:4,5:6,7:8)

microbenchmark::microbenchmark('num'={idx <- sample.int(16,1); piece <- mat.num[idx,]},
                                'df'={idx <- sample.int(16,1); piece <- df.num[idx,]})

## Unit: microseconds
## expr min lq mean median uq max neval cld
## num 4.1 5.05 7.691 6.10 7.9 38.0 100 a
## df 51.7 53.15 66.499 55.85 66.0 179.8 100 b
```

Dado que no hay un único tipo de pieza, las fichas no pueden colocarse en un tablero bidimensional. El tablero debe poder acomodar fichas que varían en cuatro atributos distintos. Ello implica que el tablero es un arreglo (*array*) tridimensional 4 x 4 x 4.

```
1 array(0, dim = c(4,4,4))
```

```
## , , 1
##
## [,1] [,2] [,3] [,4]
## [1,] 0 0 0 0
## [2,] 0 0 0 0
## [3,] 0 0 0 0
## [4,] 0 0 0 0
##
## , , 2
##
## [,1] [,2] [,3] [,4]
## [1,] 0 0 0 0
## [2,] 0 0 0 0
## [3,] 0 0 0 0
```

```

## [4,] 0 0 0 0
##
## , , 3
##
##      [,1] [,2] [,3] [,4]
## [1,] 0 0 0 0
## [2,] 0 0 0 0
## [3,] 0 0 0 0
## [4,] 0 0 0 0
##
## , , 4
##
##      [,1] [,2] [,3] [,4]
## [1,] 0 0 0 0
## [2,] 0 0 0 0
## [3,] 0 0 0 0
## [4,] 0 0 0 0

```

En cada “sub-tablero” se representa uno de los cuatro atributos de la pieza, pero la posición que ocupa la ficha es la misma siempre. Por ejemplo, en el siguiente tablero se ha colocado en la posición (2,2) la ficha (1,3,6,8)

```

## , , 1
##
##      [,1] [,2] [,3] [,4]
## [1,] 0 0 0 0
## [2,] 0 1 0 0
## [3,] 0 0 0 0
## [4,] 0 0 0 0
##
## , , 2
##
##      [,1] [,2] [,3] [,4]
## [1,] 0 0 0 0
## [2,] 0 3 0 0
## [3,] 0 0 0 0
## [4,] 0 0 0 0
##
## , , 3
##
##      [,1] [,2] [,3] [,4]
## [1,] 0 0 0 0
## [2,] 0 6 0 0
## [3,] 0 0 0 0
## [4,] 0 0 0 0
##
## , , 4
##
##      [,1] [,2] [,3] [,4]
## [1,] 0 0 0 0
## [2,] 0 8 0 0
## [3,] 0 0 0 0
## [4,] 0 0 0 0

```

## Movimientos

En QUARTO!, cualquier ficha puede ser colocada en cualquier casilla del tablero. Estos movimientos pueden lograrse simplemente asignando valores en el arreglo. Para evitar movimientos no legales y usar piezas que ya se han utilizado, pueden crearse dos objetos (matrices en este caso) que contengan las posiciones y piezas disponibles. Con cada movimiento se van actualizando.

```
1 board <- array(0,c(4,4,4))
2 # las posiciones disponibles pueden crearse del mismo modo que las piezas
3 positions <- cbind(1:4,rep(1:4,each=4)) # matriz 16 x 2
4 piezas <- create.piezas()
5
6 # elige pieza
7 pos <- positions[5,] # vector 1 x 2
8 pieza <- piezas[4,] # vector 1 x 4
9
10 # coloca pieza
11 board[pos[1],pos[2],] <- pieza
12
13 # actualiza piezas y posiciones disponibles
14 positions <- positions[-5,] # matriz 15 x 2
15 piezas <- piezas[-4,] # matriz 15 x 2
```

En la función que simula las partidas de QUARTO! no se va a encapsular el código en otra función, pero sí va a ser útil en el algoritmo minimax, por lo que a continuación se presenta la versión en C++ que se va a utilizar. Antes de ello, varios aspectos a destacar sobre las funciones en C++. Mi conocimiento sobre este lenguaje es muy limitado, por lo que tanto ésta como las demás funciones son meras “traducciones literales” de lo escrito en R. Por otro lado, como se verá frecuentemente, se está empleando de manera adicional a Rcpp, el paquete RcppArmadillo, que permite trabajar con facilidad con vectores (`vec`), matrices (`mat`) y arreglos de tres dimensiones (`cube`) (Utiliza el módulo (`template`) de C++ llamado `armadillo`).

```
1 #include <RcppArmadillo.h>
2 using namespace arma;
3 using namespace Rcpp;
4
5 // [[Rcpp::depends(RcppArmadillo)]]
6
7 // [[Rcpp::export]]
8 List Result(cube board, mat positions, int m, mat piezas, int p) {
9     rowvec pos = positions.row(m); // selecciona posición
10    rowvec pieza = piezas.row(p); // selecciona pieza
11    board.tube(pos(0)-1,pos(1)-1) = pieza; // mueve pieza a posición (índices comienzan en 0)
12    positions.shed_row(m); // actualiza posiciones
13    piezas.shed_row(p); // actualiza piezas
14    return List::create(Named("board") = board,
15                       Named("positions") = positions,
16                       Named("piezas") = piezas);
17 }
```

Conviene recordar, porque aparecerá en varios momentos, que los índices en C++ comienzan en 0. Esto significa que al utilizar un mismo objeto en ambos lenguajes, habrá que tener en cuenta la numeración de los índices si se pretende acceder a elementos concretos. En la línea 11, si se quiere acceder a la posición (1,3) en todas las dimensiones del arreglo (en R: `board[1,3,]`), hay que tener en cuenta que la primera fila es la fila 0 en C++, y que la tercera columna es la columna 2, luego las coordenadas son (0,2).

## Objetivo del juego

En esta versión de QUARTO!, al contrario que en el juego original, cada jugador elige su propia ficha y la posición donde desea colocarla. Su objetivo es establecer una línea de cuatro piezas, con al menos una característica en común. Técnicamente no tiene que establecer la línea, sino completar una, lo que quiere decir que no tiene por qué haber colocado las tres piezas anteriores.

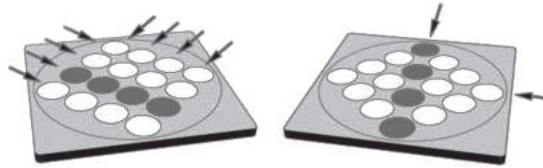


Figure 2: Una línea con fichas que comparten al menos una propiedad supone una victoria. Fuente: ultra-boardgames.com

Hay cuatro formas posibles de obtener una victoria si tenemos en cuenta que el tablero está compuesto de matrices. Al colocarse una ficha en la posición  $(i,j)$  se ha ganado si se ha completado

- la fila  $i$ ,
- la columna  $j$ ,
- la diagonal principal o
- la diagonal opuesta,

con el mismo número (misma propiedad) en al menos uno de los “sub-tableros” (uno de los atributos).

Las funciones que evalúan si se logra la victoria en alguno de los atributos son las siguientes:

```
1  #include <RcppArmadillo.h>
2  using namespace arma;
3  using namespace Rcpp;
4
5  // [[Rcpp::depends(RcppArmadillo)]]
6
7  // [[Rcpp::export]]
8  bool thatrow(mat b, int r) {
9  // Comprueba si victoria en la fila r
10
11     rowvec uniquepieces = unique(b.row(r));
12     if (sum(uniquepieces) != 0 && uniquepieces.n_elem == 1) return true;
13     return false;
14 }
15
16 // [[Rcpp::export]]
17 bool thatcol(mat b, int c) {
18 // Comprueba si victoria en la columna c
19
20     vec uniquepieces = unique(b.col(c));
```

```

21     if (sum(uniquepieces) != 0 && uniquepieces.n_elem == 1) return true;
22     return false;
23 }
24
25 // [[Rcpp::export]]
26 bool wind(mat b) {
27     // Comprueba si victoria en la diagonal principal
28
29     vec uniquepieces = unique(b.diag());
30     if (sum(uniquepieces) != 0 && uniquepieces.n_elem == 1) return true;
31     return false;
32 }
33
34 // [[Rcpp::export]]
35 bool winrevd(mat b) {
36     // Comprueba si victoria en la diagonal opuesta
37
38     vec revdiag = {b.at(3),b.at(6),b.at(9),b.at(12)}; //índices comienzan en 0
39     vec uniquepieces = unique(revdiag);
40     if (sum(uniquepieces) != 0 && uniquepieces.n_elem == 1) return true;
41     return false;
42 }

```

Todas se basan en el mismo criterio para evaluar si se ha formado una línea con el mismo número. En el caso de las filas y las columnas, se accede a los elementos de la fila/columna en la que se encuentra la ficha. Simultáneamente se extraen los valores únicos existentes en ese vector. En un “sub-tablero”, que contiene información acerca de un atributo, solo habrá tres valores únicos posibles, casilla vacía (0), propiedad 1 y propiedad 2. Si en una fila/columna solo hay un único valor, o bien la fila/columna está vacía (todo ceros), o bien se ha obtenido una victoria. Por ello, para evitar confundir filas/columnas vacías con victorias, se ha establecido el control condicional. Solo cuando haya un único elemento y no sea cero, se considera victoria. El caso para las diagonales es exactamente igual, salvo que no es necesario saber la posición de la ficha para realizar la evaluación, solo los índices que se corresponden con cada diagonal.

Estas cuatro funciones pueden integrarse en una única función que evalúa un determinado movimiento.

```

1  #include <RcppArmadillo.h>
2  using namespace arma;
3  using namespace Rcpp;
4
5  // [[Rcpp::depends(RcppArmadillo)]]
6
7  // [[Rcpp::export]]
8  int checkboard(int i, int j, cube board) {
9  // Para cada atributo, comprueba si se consigue una victoria (1).
10 // Si se finaliza el bucle, no se ha ganado (0).
11
12     for (int k = 0; k < 4; ++k) {
13         mat b = board.slice(k);
14         if (thatrow(b,i)) return 1;
15         if (thatcol(b,j)) return 1;
16         if (i == j && wind(b)) return 1;
17         if (i + j == 3 && winrevd(b)) return 1; // índices comienzan en 0
18     }
19     return 0;

```

20 }

Como puede observarse, siempre se comprueba si se da una victoria en la fila  $i$  o en la columna  $j$  cuando se coloca la ficha en  $(i,j)$ , pero solo se comprueban las diagonales cuando la ficha está en alguna de las posiciones que corresponden a las diagonales.

## Simulación de una partida (jugador aleatorio vs. jugador aleatorio)

Teniendo los elementos básicos del juego, ya es posible simular una partida entre dos jugadores, el jugador 1 y el jugador -1. Como se verá a continuación, esta identificación facilita el cambio de turno. En cualquier partida, el número máximo de jugadas (en inglés *plies*) es de 16, el número de fichas y casillas disponibles. Eso significa que cada turno (un *ply*) puede considerarse como una iteración de un bucle for. En cada iteración se produce la selección de la posición y de la ficha (ambos jugadores eligen las posiciones y las fichas aleatoriamente de las que tienen disponibles en esta primera simulación), su colocación en el tablero, la actualización del conjunto de posiciones y piezas y la evaluación de la jugada.

```
1 RAND.GAME <- function(board=array(0,c(4,4,4)),
2     player=1, # quién comienza jugando
3     positions=cbind(1:4,rep(1:4,each=4)),
4     pieces=create.pieces()){
5
6     win <- 0
7     for (ply in 1:15){
8         if (player==1){ # no es necesario un condicional
9             choice <- sample.int(17-ply,2) # porque ambos jugadores eligen igual
10        }else{ # pero si será pertinente en otras
11            choice <- sample.int(17-ply,2) # funciones
12        }
13        pos <- positions[choice[1],]
14        piece <- pieces[choice[2],]
15        positions <- positions[-choice[1],] # actualiza posiciones disponibles
16        pieces <- pieces[-choice[2],] # actualiza bolsa de fichas
17
18        board[pos[1],pos[2],] <- piece # coloca pieza en posición
19        win <- checkboard(pos[1]-1,pos[2]-1,board) # evalúa si es movimiento ganador
20        # índices comienzan en 0
21        if (win)
22            return(list(utility=player,# gana player 1 -> +1, gana player -1 -> -1
23                board=board,
24                positions_left=positions,
25                pieces_left=pieces))
26        player <- - player # cambio de turno
27    }
28    # última jugada
29    piece <- pieces
30    pos <- positions
31    board[pos[1],pos[2],] <- piece
32    win <- checkboard(pos[1]-1,pos[2]-1,board) # índices comienzan en 0
33
34    return(list(utility=ifelse(win,player,0), # win==1==TRUE -> player
35        board=board, # win==0==FALSE -> 0
36        positions_left=positions,
```



```

37     piezas_left=pieces))
38 }

```

Varios aspectos pueden destacarse de la función. Lo más notable es que el bucle solo realiza 15 iteraciones. Eso se debe a que cuando solo queda una posición o ficha disponible, el objeto deja de ser una matriz, pierde la segunda dimensión, de manera que tratar de acceder a él como si se tratase de una fila conduce a error.

```

1  positions_left <- positions[-(1:13),]
2  positions_left ; class(positions_left)

```

```

##      [,1] [,2]
## [1,]    3    4
## [2,]    4    4
## [1] "matrix" "array"

```

```

1  positions_left <- positions_left[-1,]
2  positions_left ; class(positions_left)
3
4  # positions_left[1,] >> Error in positions_left[1, ] : número incorreto de dimensiones

```

```

## [1] 4 4
## [1] "integer"

```

Por ese motivo, cuando solo queda una posición y ficha, la elección es simplemente el conjunto de posiciones y de fichas. Esta situación obliga a crear un procedimiento extra (no necesariamente una función) que evalúe el estado final del juego, pues si se produce una victoria durante el bucle for, puede simplemente asignarse una puntuación (utilidad) de 1 o -1 en función del turno. Sin embargo, al salir del bucle queda una última jugada, y tras ella puede tenerse un empate o que haya ganado el jugador -1. Con el condicional `ifelse` puede obtenerse el resultado.

Una partida de ejemplo produce un resultado como el siguiente.

```

1  set.seed(13)
2  RAND.GAME(player=1)

```

```

## $utility
## [1] -1
##
## $board
## , , 1
##
##      [,1] [,2] [,3] [,4]
## [1,]    0    2    0    0
## [2,]    0    0    0    2
## [3,]    1    1    2    1
## [4,]    0    1    0    2
##
## , , 2
##
##      [,1] [,2] [,3] [,4]
## [1,]    0    3    0    0

```

```

## [2,] 0 0 0 4
## [3,] 3 4 3 4
## [4,] 0 3 0 3
##
## , , 3
##
##      [,1] [,2] [,3] [,4]
## [1,] 0 6 0 0
## [2,] 0 0 0 5
## [3,] 5 5 5 6
## [4,] 0 6 0 6
##
## , , 4
##
##      [,1] [,2] [,3] [,4]
## [1,] 0 7 0 0
## [2,] 0 0 0 7
## [3,] 7 7 7 7
## [4,] 0 7 0 8
##
##
## $positions_left
##      [,1] [,2]
## [1,] 1 1
## [2,] 2 1
## [3,] 4 1
## [4,] 2 2
## [5,] 1 3
## [6,] 2 3
## [7,] 4 3
## [8,] 1 4
##
## $pieces_left
##      [,1] [,2] [,3] [,4]
## [1,] 1 3 5 8
## [2,] 1 3 6 8
## [3,] 1 4 5 8
## [4,] 1 4 6 8
## [5,] 2 3 5 8
## [6,] 2 4 5 8
## [7,] 2 4 6 7
## [8,] 2 4 6 8

```

En la octava jugada el jugador -1, (es el que ha empezado en segundo la partida y, por tanto, se le atribuyen las jugadas pares) hace un movimiento ganador y logra una fila de sietes.

Sabiendo todo esto, se puede proceder a realizar una primera simulación de 1000 partidas, para poder analizar posibles errores o desviaciones de lo esperado.

```

1 # las funciones escritas en C++ deben ser cargadas desde un archivo externo
2 Rcpp::sourceCpp(file=rcpp_foos)
3
4 n <- 1000
5 results <- integer(n)

```

```

6
7 set.seed(1)
8 for (i in 1:n){
9   results[i] <- RAND.GAME()$utility
10 }
11
12 table(results)

```

```

## results
## -1  0  1
## 503 21 476

```

Ambos jugadores emplean una estrategia basada en el azar, y consiguen aproximadamente la misma proporción de victorias, aunque el jugador que comienza en segundo lugar (jugador -1) obtiene una ventaja de alrededor del 3%. Un contraste sobre bondad de ajuste mediante la prueba  $X^2$  de Pearson pone de manifiesto que las proporciones encontradas no difieren de manera estadísticamente significativa de las que podrían esperarse bajo el supuesto de que es indiferente comenzar el primero o el segundo ( $P_1 = P_{-1} = 0.49$  y  $P_0 = 0.02$ )

```

1 chisq.test(table(results),p=c(.49,.02,.49))

```

```

##
## Chi-squared test for given probabilities
##
## data:  table(results)
## X-squared = 0.7949, df = 2, p-value = 0.672

```

## Estrategia de juego: Algoritmo minimax y poda alfa-beta

### Minimax

Antes de desarrollar en qué consiste el algoritmo minimax con poda alfa-beta, conviene explicar en qué consiste la estrategia minimax, pues aquel se fundamenta en éste. Toda la base teórica que se expone se ha obtenido de Russell y Norvig (2020), donde se desarrolla con mayor detalle y precisión.

De manera simplificada, minimax es un algoritmo de búsqueda especialmente diseñado para juegos con dos jugadores, de suma cero (el beneficio de un jugador es el perjuicio del otro) y con información perfecta (toda la información relativa a las posibles acciones de los jugadores, reglas... es conocida por los jugadores en todo momento), condiciones que se cumplen en este contexto. La búsqueda no es más que una exploración en profundidad, de manera recursiva, del árbol de juego, que es la estructura abstracta que representa todas las secuencias de movimientos que conducen a un estado terminal (victoria, derrota, empate, un determinado número de puntos...). Dicho de manera más informal, con la búsqueda minimax se simulan todas las partidas posibles a que da lugar cada posible movimiento en un determinado momento del juego. En la figura 1 puede verse de manera gráfica en qué consiste el árbol de juego que exploraría el algoritmo minimax.

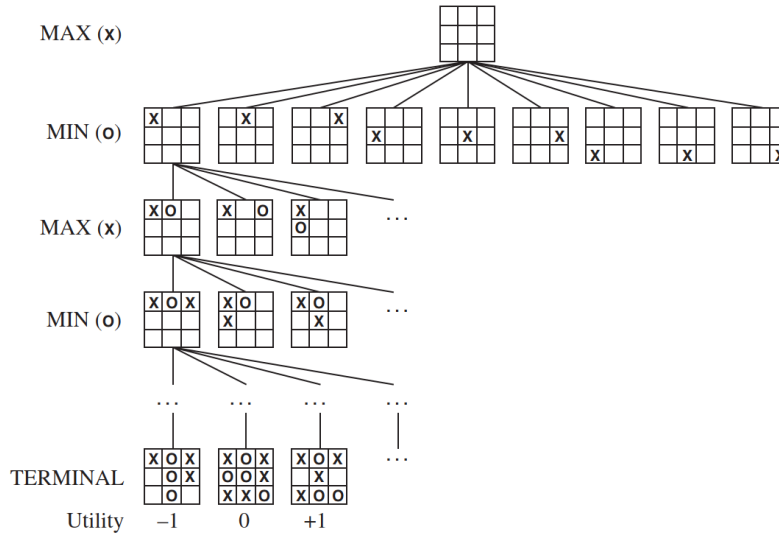


Figure 3: Árbol de juego del tres en raya. Fuente: CITA

En esta simulación se realizan tanto los propios movimientos como los del rival, y se asume que ambos juegan óptimamente. Es decir, al simular las propias acciones se seleccionan los movimientos intentando maximizar (jugador MAX) la utilidad/puntuación (+10, por ejemplo). Y cuando se simulan las acciones del contrario, se eligen también las fichas y posiciones teniendo en cuenta que su objetivo es maximizar su propia utilidad. Desde el punto de vista del propio jugador, dado que es un juego de suma cero, esto equivale a seleccionar jugadas que tratan de minimizar (jugador MIN) su utilidad (-10).

Russell y Norvig (2020) lo resumen de manera mucho más clara:

*“Given a game tree, the optimal strategy can be determined by working out the minimax value of each state in the tree, which we write as MINIMAX(s). The minimax value is the utility (for MAX) of being in that state, assuming that both players play optimally from there to the end of the game. The minimax value of a terminal state is just its utility. In a non-terminal state, MAX prefers to move to a state of maximum value when it is MAX’s turn to move, and MIN prefers a state of minimum value (that is, minimum value for MAX and thus maximum value for MIN).”*

Ahora bien, una vez se llega a un estado terminal y se calcula la utilidad, ¿cómo se “asciende” de nuevo en el árbol de juego para decidir el mejor movimiento? De nuevo, un ejemplo gráfico y la explicación de los autores resulta muy útil:

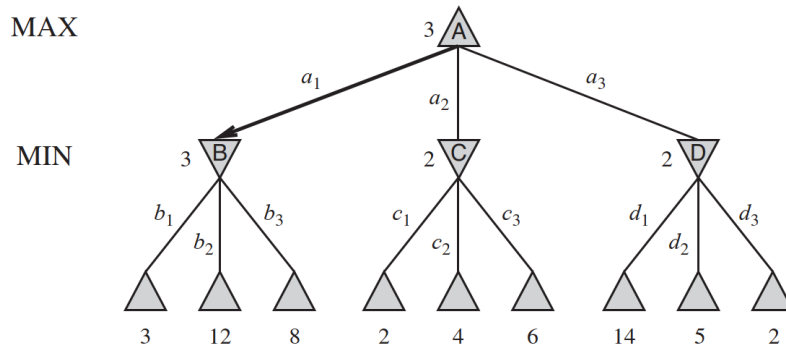


Figure 4: Selección del movimiento óptimo para MAX según la estrategia minimax: “The terminal nodes on the bottom level get their utility values from the game’s *UTILITY* function. The first MIN node, labeled B, has three successor states with values 3, 12, and 8, so its minimax value is 3. Similarly, the other two MIN nodes have minimax value 2. The root node is a MAX node; its successor states have minimax values 3, 2, and 2; so it has a minimax value of 3. We can also identify the minimax decision at the root: action  $a_1$  is the optimal choice for MAX because it leads to the state with the highest minimax value”. Fuente: Russell y Norvig (2020)

Programáticamente, el algoritmo puede implementarse haciendo uso de la recursión, estableciendo como condición de parada el llegar a un estado en el que algún jugador gane o se haya empatado. No obstante, incluso en un juego simple como el tres en raya, el árbol de juego tiene más de 360.000 nodos terminales (9!) al comenzar la partida. En el juego de QUARTO!, cuyo árbol de juego es aun mayor, para evitar tiempos de ejecución inviables, resulta conveniente fijar una condición de parada adicional, como alcanzar una determinada profundidad en el árbol de juego. Es decir, se puede considerar como estado terminal una partida no acabada si ya se han simulado, por ejemplo, 4 turnos (*plies*).

La gran ventaja del algoritmo minimax sobre otras posibles estrategias, es que garantiza la selección óptima de movimientos. Contra un oponente que también juega óptimamente, las partidas deberían acabar en empate. No obstante, esto no ocurre si la profundidad de búsqueda se limita, como es el caso. De todas formas, al enfrentarse a jugadores subóptimos, esta limitación puede verse compensada.

### Poda alfa-beta

Una alternativa para mejorar la eficiencia de minimax es el algoritmo de poda alfa-beta. Esta técnica permite “podar”, dejar sin explorar, ramas del árbol de juego que no suponen ninguna diferencia porque no conducen a resultados mejores que los ya explorados. Al reducirse el tiempo de búsqueda, se pueden “reinvertir” esos recursos en aumentar la profundidad de búsqueda.

En el algoritmo, alfa es el valor del mejor movimiento encontrado hasta el momento para el jugador MAX (máxima utilidad). Y beta es el valor del mejor movimiento encontrado hasta el momento para el jugador MIN (mínima utilidad). Estos valores son los que permiten interrumpir la búsqueda cuando es necesario (finalizar las llamadas recursivas). En todo momento, el jugador MAX seleccionará como mejor movimiento aquel que como mínimo dé un valor de alfa, y el jugador MIN elegirá siempre el movimiento que como máximo le reporte una utilidad de beta.

Me remito a Russell y Norvig (2020) para una explicación paso a paso del funcionamiento de la poda alfa-beta:

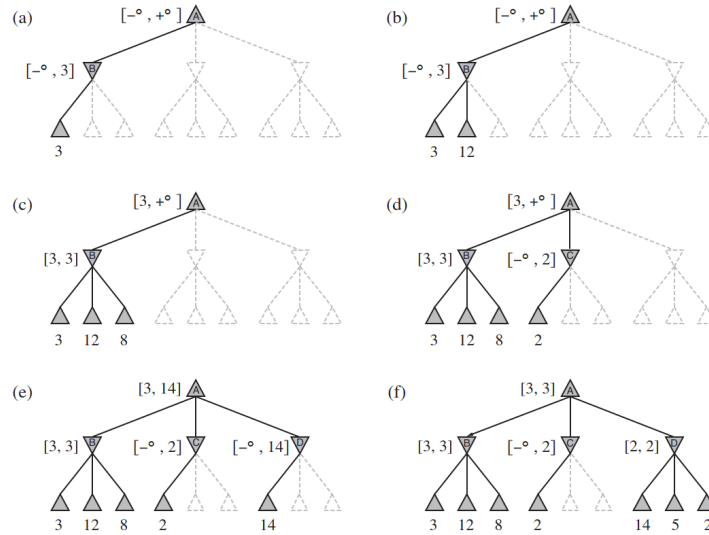


Figure 5: Poda alfa-beta: "At each point, we show the range of possible values for each node. (a) The first leaf below B has the value 3. Hence, B, which is a MIN node, has a value of at most 3. (b) The second leaf below B has a value of 12; MIN would avoid this move, so the value of B is still at most 3. (c) The third leaf below B has a value of 8; we have seen all B's successor states, so the value of B is exactly 3. Now we can infer that the value of the root is at least 3, because MAX has a choice worth 3 at the root. (d) The first leaf below C has the value 2. Hence, C, which is a MIN node, has a value of at most 2. But we know that B is worth 3, so MAX would never choose C. Therefore, there is no point in looking at the other successor states of C. This is an example of alpha-beta pruning. (e) The first leaf below D has the value 14, so is worth at most 14. This is still higher than MAX's best alternative (i.e., 3), so we need to keep exploring D's successor states. Notice also that we now have bounds on all of the successors of the root, so the root's value is also at most 14. (f) The second successor of D is worth 5, so again we need to keep exploring. The third successor is worth 2, so now D is worth exactly 2. MAX's decision at the root is to move to B, giving a value of 3. Fuente: Russell y Norvig (2020)

### Pseudocódigo y código C++

La implementación del algoritmo alfa-beta minimax se ha basado casi exclusivamente en dos fuentes, Russell y Norvig (2020) y las [diapositivas](#) del profesor del Worcester Polytechnic Institute, Charles Rich.

## Minimax Algorithm

```

def MinMax (board, player, depth, maxDepth)
  if ( board.isGameOver() or depth == maxDepth )
    return board.evaluate(player), null

  bestMove = null
  if ( board.currentPlayer() == player )
    bestScore = -INFINITY
  else bestScore = +INFINITY
  Note: makeMove returns copy of board
  (can also move/unmove--but don't execute graphics!)

  for move in board.getMoves()
    newBoard = board.makeMove(move)
    score = MinMax(newBoard, player, depth+1, maxDepth)
    if ( board.currentPlayer() == player )
      if ( score > bestScore ) # max
        bestScore = score
        bestMove = move
      Note: test works for multiplayer
    else
      if ( score < bestScore ) # min
        bestScore = score
        bestMove = move
      case also

  return bestScore, bestMove

MinMax(board, player, 0, maxDepth)

```



 IMGD 4000 (D 09) 12

Figure 6: Pseudocódigo del algoritmo minimax. Fuente: diapositivas profesor Rich

```

function ALPHA-BETA-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state, -∞, +∞)
  return move

function MAX-VALUE(game, state, α, β) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← -∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a), α, β)
    if v2 > v then
      v, move ← v2, a
      α ← MAX(α, v)
    if v ≥ β then return v, move
  return v, move

function MIN-VALUE(game, state, α, β) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a), α, β)
    if v2 < v then
      v, move ← v2, a
      β ← MIN(β, v)
    if v ≤ α then return v, move
  return v, move

```

Figure 7: Pseudocódigo del algoritmo minimax con poda alfa-beta. Fuente: Russell y Norvig (2020)

El programa finalmente diseñado incorpora aspectos de ambos planteamientos. Por ejemplo, en vez de crear una función MAX\_VALUE para el jugador MAX y una función MIN\_VALUE para el jugador MIN, se ha establecido un control condicional en la misma función AB\_MINIMAX para controlar si es el turno de uno u otro jugador, como en la diapositiva del profesor Rich. El código en C++ es el siguiente:

```

1  #include <RcppArmadillo.h>
2  using namespace arma;
3  using namespace Rcpp;
4
5  // [[Rcpp::depends(RcppArmadillo)]]
6
7  // [[Rcpp::export]]
8  List AB_MINIMAX(cube board, int ply, int player,

```

```

9         mat positions, mat pieces,
10         int depth=0, int Alpha=-100, int Beta=100) {
11
12 // Obtiene el mejor movimiento basándose en la estrategia minimax
13 // con poda alfa-beta.
14
15 List best = List::create(Named("value") = R_NilValue,
16                          Named("pos") = R_NilValue,
17                          Named("piece") = R_NilValue);
18
19 bool terminal_state = is_terminal(board,ply+depth);
20 if (terminal_state || depth == 4) {
21     int v = payoff(terminal_state,player,depth);
22     best["value"] = v;
23     return best;
24 }
25
26 if (player == 1) { // jugador MAX
27     int v = -100;
28     int pos_rows = positions.n_rows;
29     int piece_rows = pieces.n_rows;
30     for (int m = 0; m < pos_rows; ++m) {
31         for (int p = 0; p < piece_rows; ++p) {
32             List upd = Result(board,positions,m,pieces,p); // actualiza partida
33
34             // ejecuta de nuevo minimax con estado de la partida actualizado,
35             // siendo el turno del jugador MIN
36             List output = AB_MINIMAX(upd["board"],ply,-player,
37                                     upd["positions"],upd["pieces"],
38                                     depth+1,Alpha,Beta);
39
40             int mmx_value = output["value"];
41             if (mmx_value > v) { // si punt. del movimiento es mejor que el del
42                 v = mmx_value; // movimiento anterior, modifica la puntuación
43                 best["value"] = mmx_value;
44                 best["pos"] = positions.row(m); // y modifica movimiento asociado
45                 best["piece"] = pieces.row(p); // a esa puntuación
46                 Alpha = std::max(Alpha,v); // actualiza punt. mínima aceptable
47             }
48             if (v >= Beta) return best;
49         }
50     }
51 }
52 }else { // jugador MIN
53     int v = 100;
54     int pos_rows = positions.n_rows;
55     int piece_rows = pieces.n_rows;
56     for (int m = 0; m < pos_rows; ++m) {
57         for (int p = 0; p < piece_rows; ++p) {
58             List upd = Result(board,positions,m,pieces,p);
59
60             // ejecuta de nuevo minimax con estado de la partida actualizado,
61             // siendo el turno del jugador MAX

```



```

62     List output = AB_MINIMAX(upd["board"],ply,-player,
63                               upd["positions"],upd["pieces"],
64                               depth+1,Alpha,Beta);
65
66     int mmx_value = output["value"];
67     if (mmx_value < v) {
68         v = mmx_value;
69         best["value"] = mmx_value;
70         best["pos"] = positions.row(m);
71         best["piece"] = pieces.row(p);
72         Beta = std::min(Beta,v); // actualiza punt. máxima aceptable
73     }
74     if (v <= Alpha) return best;
75 }
76 }
77 }
78
79 return best;
80 }

```

### El programa, paso a paso

Para explicar el funcionamiento paso a paso del programa, merece la pena volver de nuevo a la figura 8.

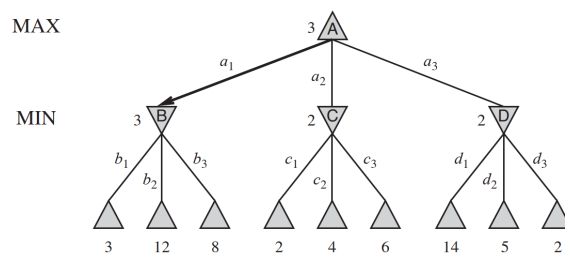


Figure 8: Árbol de juego

Comienza siendo el turno del jugador MAX, y podemos suponer que es uno de los últimos *ply*, porque solo tiene disponibles tres posibles acciones. Al llamar a la función `AB_MINIMAX` lo primero que se hace es crear el objeto que contendrá el movimiento elegido. Luego se evalúa el estado de la partida. Si se ha programado bien, nunca debe concluirse que la partida ha finalizado en la primera llamada, porque en ese caso ni siquiera tendría sentido que el jugador estuviese eligiendo un movimiento. La función que comprueba si nos encontramos en un estado terminal es similar a la función que comprueba movimientos ganadores, con la diferencia de que no solo se comprueba una única fila o una única columna, sino todas. Se utilizan números enteros en vez de valores lógicos porque para calcular posteriormente la puntuación obtenida, conviene distinguir entre una partida acabada por victoria/derrota o un empate.

```

1  #include <RcppArmadillo.h>
2  using namespace arma;
3  using namespace Rcpp;
4
5  // [[Rcpp::depends(RcppArmadillo)]]

```

```

6
7 // [[Rcpp::export]]
8 bool anyrow(mat b) {
9 // Comprueba si se consigue victoria en alguna de las filas
10
11     for (int r = 0; r<4; ++r) {
12         if (sum(b.row(r)) == 0) continue;
13         rowvec uniquepieces = unique(b.row(r));
14         if (uniquepieces.n_elem == 1) return true;
15     }
16     return false;
17 }
18
19 // [[Rcpp::export]]
20 bool anycol(mat b) {
21 // Comprueba si se consigue victoria en alguna de las columnas
22
23     for (int c = 0; c<4; ++c) {
24         if (sum(b.col(c)) == 0) continue;
25         vec uniquepieces = unique(b.col(c));
26         if (uniquepieces.n_elem == 1) return true;
27     }
28     return false;
29 }
30
31 // [[Rcpp::export]]
32 int is_terminal(cube board, int ply) {
33 // Determina si la partida ha acabado con victoria de algún jugador (1),
34 // con empate (2), o si aun no ha finalizado (0)
35
36     for (int k=0; k<4; ++k) {
37         mat b = board.slice(k);
38         if (anyrow(b)) return 1;
39         if (anycol(b)) return 1;
40         if (wind(b)) return 1;
41         if (winrevd(b)) return 1;
42     }
43     if (ply == 16) return 2; // empate
44     return 0;
45 }

```

Como no se ha acabado la partida, se continúa. Es el turno del jugador 1 (jugador MAX), por lo que se ejecuta el primer bloque (línea 26). Mediante el uso de bucles anidados se recorren todos los movimientos disponibles. El primer movimiento en la figura es  $a_1$  y con la función `Result` que hemos visto anteriormente simplemente se actualiza el estado de la partida. Se coloca la ficha de  $a_1$  en la posición de  $a_1$  y se eliminan esa ficha y posición de las disponibles. Este nuevo estado de la partida es el argumento de la función `AB_MINIMAX` que se llama de nuevo. Ahí comienza el proceso recursivo de simulación de partidas. Con la nueva llamada se vuelve a examinar el tablero. Aun no ha acabado la partida. Ahora es el turno del jugador -1 (jugador MIN). De nuevo, empieza a recorrer todas las acciones disponibles, empezando por  $b_1$ . Y una vez hecho el movimiento, otra vez se vuelve a iniciar el algoritmo minimax.

Esta vez, el movimiento sí conduce a un estado terminal y reporta una utilidad de 3, que es calculada con la función `payoff`. No se denomina `utility` porque es algo más compleja que una función que devuelve un +1 con una victoria del jugador 1, un -1 con una victoria del jugador -1 o un 0 con un empate. Como puede

verse debajo, la puntuación otorgada varía en función de la profundidad. Victorias que se logran en más movimientos son penalizadas. Hay que tener en cuenta, además, que la función está evaluando el tablero tal y como es después del movimiento del jugador anterior, puesto que cuando se llama a la función ya se ha cambiado el turno. De ahí el argumento `next_player`.

```

1  #include <RcppArmadillo.h>
2  using namespace arma;
3  using namespace Rcpp;
4
5  // [[Rcpp::depends(RcppArmadillo)]]
6
7  // [[Rcpp::export]]
8  int payoff(int winstate, int next_player, int depth) {
9  // Calcula puntuación obtenida al llegar a un estado terminal
10
11     if (winstate == 1) {
12         if (next_player == -1) return 10 - depth;
13         return -10 + depth;
14     }
15     return 0;
16 }

```

Suponemos que la secuencia  $a_1-b_1$  tiene una puntuación de 3 (aunque la función `payoff` realmente no podría devolver ese valor). El algoritmo entonces devuelve una lista con ese valor y con ficha y posición nulos. Este es el resultado de la variable `output` en la segunda “capa”. Si se continúa la función (línea 66) se observa que se extrae el valor de 3 y se compara con el valor  $v$  que se tenía anteriormente, y que inicialmente es un número muy alto (para el jugador MIN). Obviamente 3 es menor que 100, y por tanto es la mejor puntuación encontrada hasta el momento, y la posición y pieza asociadas también son las mejores hasta el momento. Además, se actualiza el criterio de peor valor posible. El nuevo valor de **Beta** es  $v$ , es decir, 3. El jugador MIN elegirá un movimiento que como máximo le dé una puntuación de 3. La comparación entre la puntuación encontrada y **Alfa** no produce de momento ningún resultado porque **Alfa** es muy elevado. Tras esta actualización se comienza la segunda iteración del bucle, que en la figura corresponde a examinar la jugada  $b_2$ . Luego se examinaría el movimiento  $b_3$ . Se realiza el mismo proceso de antes y se va actualizando el valor de **Beta**, pero se mantiene en 3 porque es el mínimo (como se puede ver en la figura 9).

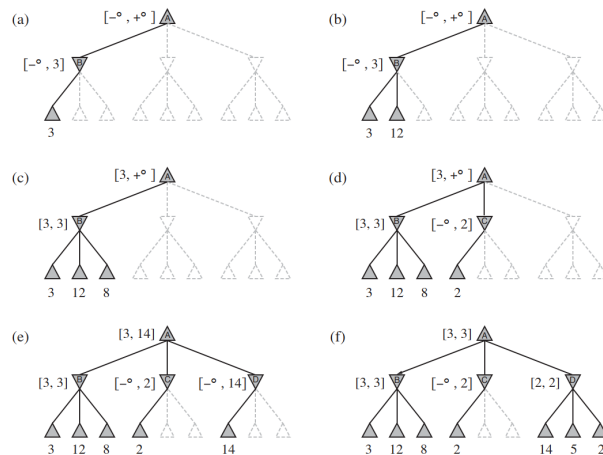


Figure 9: Poda alfa-beta

Habiendo examinado todos los movimientos del jugador MIN, se “retrocede” un turno, porque ha finalizado la segunda de las llamadas recursivas (línea 73). Es entonces el turno del jugador MAX, que estaba examinando el movimiento  $a_1$ . Ahora ya sabemos que su valor es de 3. Como es mayor que el valor  $v$  por defecto (-100), el movimiento asociado es el mejor, de momento. Además, en la línea 43 vemos que, como ocurría con el jugador MIN, el mejor valor aceptable (mínimo aceptable en este caso) se modifica. Ahora Alfa vale 3. Continuando con la ejecución, vemos que llegamos a la comparación entre  $v$  y Beta. Es importante recordar que Beta no vale 3, sino 100, porque su valor ha cambiado en otra llamada recursiva, no aquí. Por tanto, se continúa el bucle que examina movimientos. El siguiente por explorar es el movimiento  $a_2$ . En la línea 33 se vuelve a llamar a AB\_MINIMAX, como ha ocurrido antes. Vuelve a ser el turno de MIN.

Al llegar al nodo terminal se obtiene una puntuación de 2, que se convierte en la mejor puntuación hasta el momento para el movimiento que hace MIN como respuesta a  $a_2$ . Beta se convierte en 2, y llegamos de nuevo al condicional de la línea 68. Pero esta vez Alfa vale 3, que es el mínimo aceptable que se había obtenido tras explorar la primera rama del árbol. Y en este caso, el valor de  $v$  es 2, por lo que directamente se devuelve el movimiento obtenido. El resto de movimientos,  $c_2$  y  $c_3$  no tienen que ser explorados, se han podado esas ramas. ¿Por qué? Como bien explican Russell y Norvig (2020), al estar en el turno de MIN y haberse hallado un valor Beta (puntuación máxima aceptable) de 2, si MIN sigue explorando  $c_2$  y  $c_3$  elegirá como mejor movimiento aquel que dé una puntuación de 2 ó menos. Eso significa que al terminar la llamada recursiva y ascender en el árbol al nodo  $a_2$ , llegará una puntuación de 2 ó menos. Pero al ser el turno de MAX, cuando tenga que elegir entre  $a_1$ ,  $a_2$  y  $a_3$ , sabemos que nunca elegirá  $a_2$ , porque la puntuación mínima aceptable se ha fijado en 3 (valor del movimiento  $a_1$ ). Por eso es posible descartar directamente esa rama y dejar sin explorar sus ramificaciones restantes.

Con el resto de movimientos ( $a_3$  y sus correspondientes llamadas recursivas para explorar  $d_1$ ,  $d_2$  y  $d_3$ ) ocurre lo mismo que se ha comentado en los párrafos anteriores.

Para ver el algoritmo en acción podemos simular una partida e interrumpirla en la jugada 12, por ejemplo. Es el turno 13, por tanto, mueve el jugador 1, que vamos a suponer que es el que emplea el algoritmo.

```
RAND.GAME_LIMITED <- function(board=array(0,c(4,4,4)),
                               player=1,limit,
                               positions=cbind(1:4,rep(1:4,each=4)),
                               pieces=create.pieces()){

  win <- 0
  for (ply in 1:limit){
    if (player==1){
      choice <- sample.int(17-ply,2)
    }else{
      choice <- sample.int(17-ply,2)
    }
    pos <- positions[choice[1],]
    piece <- pieces[choice[2],]
    positions <- positions[-choice[1],]
    pieces <- pieces[-choice[2],]

    board[pos[1],pos[2],] <- piece
    win <- checkboard(pos[1]-1,pos[2]-1,board)

    if (win) break # si se produce una victoria sal del bucle
    player <- - player
  }

  # al finalizar el bucle simplemente devuelve el estado de la partida
  return(list(utility=ifelse(win,player,0),
             board=board,
```

```

        positions_left=positions,
        pieces_left=pieces))
}

```

```

set.seed(41)
( demo <- RAND.GAME_LIMITED(limit=12) )

```

```

## $utility
## [1] 0
##
## $board
## , , 1
##
##      [,1] [,2] [,3] [,4]
## [1,]    2    2    1    0
## [2,]    2    0    2    0
## [3,]    1    1    1    0
## [4,]    2    1    2    1
##
## , , 2
##
##      [,1] [,2] [,3] [,4]
## [1,]    3    4    4    0
## [2,]    3    0    3    0
## [3,]    4    3    4    0
## [4,]    3    3    4    4
##
## , , 3
##
##      [,1] [,2] [,3] [,4]
## [1,]    5    6    5    0
## [2,]    5    0    6    0
## [3,]    6    5    5    0
## [4,]    6    6    5    6
##
## , , 4
##
##      [,1] [,2] [,3] [,4]
## [1,]    8    7    7    0
## [2,]    7    0    7    0
## [3,]    8    8    8    0
## [4,]    8    7    8    7
##
##
## $positions_left
##      [,1] [,2]
## [1,]    2    2
## [2,]    1    4
## [3,]    2    4
## [4,]    3    4
##
## $pieces_left
##      [,1] [,2] [,3] [,4]

```

```
## [1,] 1 3 5 7
## [2,] 1 3 6 8
## [3,] 2 4 5 7
## [4,] 2 4 6 8
```

Varias posiciones conducen a victoria. Se puede colocar la ficha (1,3,5,7) o la (1,3,6,8) en la posición (1,4), o se pueden colocar las piezas (1,3,6,8) o (2,4,6,8) en la posición (3,4). Dado que la posición (1,4) y la ficha (1,3,5,7) se examinan antes, éste par posición-pieza debería ser la elección del algoritmo.

```
AB_MINIMAX(demo$board,ply=13,player=1,demo$positions_left,demo$pieces_left)
```

```
## $value
## [1] 9
##
## $pos
## [1] 1 4
##
## $piece
## [1] 1 3 5 7
```

## Simulación de una partida (alfa-beta minimax vs. jugador aleatorio)

Para permitir que el agente que implementa la estrategia minimax juegue, es necesario modificar la función `RAND.GAME`. La selección del movimiento para uno de los jugadores ya no es azarosa. Sin embargo, técnicamente es posible que la estrategia no permita seleccionar una acción que sea superior a las demás. Puede darse la situación de que todos los movimientos conlleven la misma puntuación, quizá porque la derrota sea inminente, por ejemplo. En tal caso, la lista que devuelve `AB_MINIMAX` es la creada por defecto, con valores nulos. Si eso ocurre, cualquier acción vale como mejor acción y puede seleccionarse al azar entre las disponibles. En los demás casos, la función elegirá una posición y una ficha. El problema es que para actualizar correctamente la “bolsa” de fichas y de posiciones, se requiere el índice que identifica la pieza y la posición, no los elementos en sí. Para obtener este índice a partir del valor de la pieza y posición seleccionadas se emplea la función `selected`, cuyo código se ha obtenido de este [hilo](#) de Stackoverflow.

```
1 selected <- function(action,actions){
2   # devuelve número de fila que corresponde al vector action en la matriz actions
3
4   which(rowSums(actions == action[col(actions)]) == ncol(actions))
5 }
```

La función completa es la siguiente:

```
1 GAME <- function(board=array(0,c(4,4,4)),
2   player=1,
3   positions=cbind(1:4,rep(1:4,each=4)),
4   pieces=create.pieces()){
5
6   win <- 0
7   for (ply in 1:15){
8     if (player==1){ # jugador 1 utiliza alfa-beta Minimax
9       mmax <- AB_MINIMAX(board,ply,player,positions,pieces)
10      if(is.null(mmax$pos)){ # si no se encuentra movimiento óptimo elige al azar
11        choice <- sample.int(17-ply,2)
```

```

12     }else{
13         choice <- c(selected(mmax$pos,positions),selected(mmax$piece,pieces))
14     }
15     }else{
16         choice <- sample.int(17-ply,2)
17     }
18     pos <- positions[choice[1],]
19     piece <- pieces[choice[2],]
20     positions <- positions[-choice[1],]
21     pieces <- pieces[-choice[2],]
22
23     board[pos[1],pos[2],] <- piece
24     win <- checkboard(pos[1]-1,pos[2]-1,board)
25
26     if (win)
27         return(list(utility=player,
28                     board=board,
29                     positions_left=positions,
30                     pieces_left=pieces))
31     player <- - player
32 }
33
34 piece <- pieces
35 pos <- positions
36 board[pos[1],pos[2],] <- piece
37 win <- checkboard(pos[1]-1,pos[2]-1,board)
38
39 return(list(utility=ifelse(win,player,0),
40             board=board,
41             positions_left=positions,
42             pieces_left=pieces))
43 }

```

Podemos comprobar la eficacia de la estrategia alfa-beta minimax simulando 100 partidas. El número de simulaciones se ha visto reducido debido a que el tiempo de ejecución es ahora mucho mayor. Además, el hecho de emplear código de C++ dificulta la paralelización (ver, por ejemplo, este [hilo](#) de stackoverflow). A pesar de todo, la eficiencia es notablemente superior a la que se obtiene programando directamente en R (ver apéndice).

```

# las funciones escritas en C++ ya han sido cargadas anteriormente
# Rcpp::sourceCpp(rcpp_foos)

n <- 50

results_mmx_first <- integer(n)
results_mmx_second <- integer(n)

set.seed(666)
for (i in 1:n){
    results_mmx_first[i] <- GAME(player=1)$utility # comienza minimax
    results_mmx_second[i] <- GAME(player=-1)$utility # comienza bot
}

```

```
table(results_mmx_first) ; table(results_mmx_second)
```

```
## results_mmx_first
## 1
## 50
## results_mmx_second
## 0 1
## 1 49
```

Puede observarse de manera clara que alfa-beta minimax es absolutamente superior a la estrategia basada en el azar. Además, resulta irrelevante que se comience jugando o que se empiece con el segundo turno. Pero la estrategia no es infalible, pues incluso ante el azar puede empatar. Esto se debe a que el comportamiento del algoritmo está guiado por el supuesto de rival racional. Bajo la asunción de que el oponente trata de maximizar su propia utilidad escogiendo las mejores jugadas posibles, el jugador minimax debe evitar ciertos movimientos durante la partida, como colocar la tercera pieza con una misma propiedad en una fila o columna. Esta “prudencia” añadida ante un jugador aleatorio no es un inconveniente en la mayoría de casos, pero en situaciones excepcionales (una de cada cincuenta aproximadamente) puede resultar perjudicial porque se van descartando potenciales movimientos ganadores, y si el *bot* no comete errores (por puro azar, obviamente), se acaban las fichas.

### Simulación de una partida (alfa-beta minimax vs. humano)

Finalmente, también es posible editar la función `GAME` para que uno de los jugadores sea una persona, y no un *bot*. Aunque la función falla en proporcionar una buena experiencia de usuario, permite interactuar eficazmente con el sistema. De todas formas, para evitar que el jugador humano tenga que utilizar el tablero tridimensional y las piezas y posiciones en forma de matriz, se han creado un par de funciones que presentan el estado de la partida de forma más atractiva.

```
show.positions <- function(positions){
# convierte la matriz de posiciones disponibles en un único vector de caracteres
# y proporciona apariencia de coordenadas a las posiciones

  apply(positions,1,function(r) paste0('(',r[1],',',r[2],')',collapse=''))
}

show.pieces <- function(pieces){
# convierte matriz de piezas disponibles en un único vector de caracteres

  apply(pieces,1,function(r) paste0(r,collapse=''))
}

print(noquote(show.positions(demo$positions_left)))
```

```
## [1] (2,2) (1,4) (2,4) (3,4)
```

```
print(noquote(show.pieces(demo$pieces_left)))
```

```
## [1] 1357 1368 2457 2468
```

En la nueva función, se incluyen estas funciones y además un tablero bidimensional en el que se van mostrando los movimientos.



```

1 GAME.INTERACTIVE <- function(board=array(0,c(4,4,4)),
2     player=1,
3     positions=cbind(1:4,rep(1:4,each=4)),
4     pieces=create.pieces()){
5
6     board2D <- matrix(0,4,4)
7     win <- 0
8     for (ply in 1:15){
9         positions2D <- show.positions(positions)
10        pieces2D <- show.pieces(pieces)
11
12        if (player==1){ # jugador 1 utiliza alfa-beta Minimax
13            mmax <- AB_MINIMAX(board,ply,player,positions,pieces)
14            if(is.null(mmax$pos)){
15                choice <- sample.int(17-ply,2)
16            }else{
17                choice <- c(selected(mmax$pos,positions),selected(mmax$piece,pieces))
18            }
19        }else{ # jugador humano
20            # muestra estado de la partida
21            print(noquote(cbind(positions2D)))
22            print(noquote(cbind(pieces2D)))
23            print(noquote(board2D))
24
25            x <- menu(1:(17-ply),title='movimiento') # posiciones disponibles
26            y <- menu(1:(17-ply),title='pieza') # piezas disponibles
27            choice <- c(x,y)
28        }
29        pos <- positions[choice[1],]
30        piece <- pieces[choice[2],]
31        positions <- positions[-choice[1],]
32        pieces <- pieces[-choice[2],]
33
34        board[pos[1],pos[2],] <- piece
35        board2D[pos[1],pos[2]] <- pieces2D[choice[2]] # actualiza tablero 2D
36        win <- checkboard(pos[1]-1,pos[2]-1,board)
37
38        if (win)
39            return(list(winner=player,
40                        board=noquote(board2D)))
41        player <- - player
42    }
43
44    piece <- pieces
45    pos <- positions
46    board[pos[1],pos[2],] <- piece
47    board2D[pos[1],pos[2]] <- show.pieces(piece) # actualiza tablero 2D
48    win <- checkboard(pos[1]-1,pos[2]-1,board)
49
50    return(list(winner=ifelse(win,player,0),
51              board=noquote(board2D)))
52 }

```

Cuando es el turno del jugador humano se le presenta el estado actual de la partida, el tablero, las posiciones y las piezas disponibles. Primero debe elegir una posición y luego una ficha. El número seleccionado debe corresponder con el número de fila de la posición y de la ficha.

Tras jugar varias partidas se puede hacer una valoración cualitativa de la experiencia. No es imposible vencer en el juego, a pesar de que el algoritmo puede explorar muchísimas más opciones que yo. Sin embargo, explota con mucha eficacia las confusiones y momentos de falta de concentración. Es posible que una combinación de tres propiedades iguales en una columna pase desapercibida para el jugador humano (me ha ocurrido en un par de ocasiones, como en el ejemplo grabado que se puede encontrar en la carpeta de materiales), pero estos deslices naturalmente no le suceden al “jugador minimax”. Por otra parte, parece haber diferencias en lo referido al turno de cada jugador. Cuando he tenido que comenzar la partida en segundo lugar (turnos pares), he tenido la impresión (no ha habido ningún tipo de análisis ni registro de datos) de tener cierta ventaja, quizá porque las líneas que se tienen que formar son de cuatro piezas y es más fácil forzar al contrario a realizar movimientos peores.

## Comentarios y conclusiones

La implementación del algoritmo es aparentemente exitosa y no parece presentar errores. El análisis de partidas movimiento a movimiento revela que el agente se comporta de acuerdo con lo esperado. Por ejemplo, cuando en una partida se tienen dos fichas alineadas y se tiene la posibilidad de formar una línea de tres, el agente lo evita, dado que el algoritmo se basa en el supuesto de que ambos jugadores juegan óptimamente y que, por tanto, en el siguiente turno el rival completaría esa línea.

Desde el punto de vista del rendimiento en partidas con jugadores humanos, hay varios aspectos a destacar. El algoritmo no es imbatible. Ya se ha comentado que la limitación de la profundidad de búsqueda impide simular partidas completas para obtener puntuación de ellas, por lo que no siempre se obtiene el movimiento óptimo. Esta restricción, que es prácticamente inevitable si se quiere garantizar un tiempo de ejecución razonable, puede verse relativamente compensada con una buena función de evaluación (función *payoff*). El mayor beneficio del algoritmo de poda alfa-beta se produce cuando se exploran primero los mejores movimientos, porque así se podan antes más ramas del árbol. Esto exige una función *payoff* que puntúe los diferentes estados “pseudo-terminales” de manera más compleja. La función aquí utilizada proporciona una puntuación de 0 a las partidas que no se han terminado, sin hacer distinciones. La única discriminación realizada en este caso ha sido entre las jugadas ganadoras, las que se obtienen más tarde son penalizadas. Sin embargo, una partida no acabada en la que hay, por ejemplo, dos líneas con dos fichas con la misma propiedad, podría ser mejor puntuada que una en la que no hay líneas empezadas. Esta discriminación podría favorecer un mejor ordenamiento de los movimientos y, por tanto, no solo una mejor selección, sino una búsqueda más eficiente, porque se podrían dejar sin explorar otros caminos.

Desde una perspectiva psicológica, no obstante, estas limitaciones resultan interesantes. El algoritmo está emulando a un jugador que, lógicamente, busca su máximo beneficio, pero que tiene en cuenta que el otro jugador también. Es capaz de anticipar las respuestas que hará en función de sus movimientos, y elige basándose en esas simulaciones “mentales”. Además, su capacidad de previsión “solo” alcanza cuatro turnos o *plies*, existe una limitación en la memoria de trabajo. Por supuesto, las semejanzas son bastante superficiales. La selección de movimientos en jugadores humanos no se basa, al menos en la mayoría de ocasiones, y sobre todo en juegos complejos, en el cálculo de utilidades asociadas a distintas secuencias de movimientos. Y mucho menos siguiendo unas reglas estrictas de asignación de puntuaciones. Por otro lado, los jugadores humanos suelen ser mucho más eficientes, en tanto que no es común realizar una búsqueda exhaustiva entre el conjunto de movimientos disponibles. Se suelen emplear heurísticos que acortan y limitan la búsqueda a aquellos movimientos potencialmente buenos. Por último, otra diferencia está en que los jugadores humanos pueden aprender de partidas anteriores. El agente aquí diseñado no. El algoritmo minimax es determinista en el sentido de que ante una misma secuencia de jugadas siempre responderá con el mismo movimiento. Ello implica que, por ejemplo, ante un jugador que emplea el azar como estrategia, siempre necesitará aproximadamente el mismo número de partidas para ganar, aun cuando simplemente tratando de alinear cuatro fichas en los cuatro primeros turnos es prácticamente garantía de victoria.

## Apéndice: comparación R y C++

Inicialmente, todas las funciones aquí presentadas se programaron en R. Sin embargo, al ejecutar algunas simulaciones de partidas se comprobó que el tiempo de ejecución resultaba excesivo. Es por ello que se decidió implementar las funciones más utilizadas y aquellas que emplean bucles o llamadas recursivas en C++, a través de la interfaz que proporciona el paquete Rcpp. La ganancia en términos de rendimiento se observó en todas las funciones. Baste ver el ejemplo de la función AB\_MINIMAX (y sus funciones auxiliares).

```
# Equivalencias en R

Result_R <- function(board,positions,m,pieces,p){
  pos <- positions[m,]
  piece <- pieces[p,]
  board[pos[1], pos[2],] <- piece
  return(list(board=board,
              positions=positions[-m,],
              pieces=pieces[-p,]))
}

anyrow_R <- function(b){
  for (r in 1:nrow(b)){
    if(sum(b[r,]) == 0) next
    if(length(unique(b[r,])) == 1) return(TRUE)
  }
  return(FALSE)
}

anycol_R <- function(b){
  for (c in 1:ncol(b)){
    if(sum(b[,c]) == 0) next
    if(length(unique(b[,c])) == 1) return(TRUE)
  }
  return(FALSE)
}

wind_R <- function(b){
  uniquepieces <- unique(diag(b))
  if(sum(uniquepieces) != 0 && length(uniquepieces) == 1) return(TRUE)
  return(FALSE)
}

winrevd_R <- function(b){
  uniquepieces <- unique(b[c(4,7,10,13)])
  if(sum(uniquepieces) != 0 && length(uniquepieces) == 1) return(TRUE)
  return(FALSE)
}

is_terminal_R <- function(board,ply){
  for (k in 1:4){
    b <- board[, ,k]
    if (anyrow_R(b)) return(1)
    if (anycol_R(b)) return(1)
    if (wind_R(b)) return(1)
    if (winrevd_R(b)) return(1)
  }
}
```

```

}
if (ply == 16+1) return(2) # empate al acabar la partida
return(0)
}

payoff_R <- function(winstage,next_player,depth){
  if(winstage == 1){
    if(next_player == -1) return(10 - depth)
    return(-10 + depth)
  }
  return(0)
}

AB_MINIMAX_R <- function(board,ply,player,positions,pieces,depth=0,Alpha=-Inf,Beta=Inf){

  best <- list(value=NULL,pos=NULL,piece=NULL)
  terminal.state <- is_terminal_R(board,ply+depth)
  if (terminal.state || depth==4){
    best$value <- payoff_R(terminal.state,player,depth)
    return(best)
  }

  if (ply+depth==16){ # requerido cuando se trata del último turno
    positions <- rbind(positions)
    pieces <- rbind(pieces)
  }

  if (player==1){
    v <- -Inf
    for (m in 1:nrow(positions)){
      for (p in 1:nrow(pieces)){
        upd <- Result_R(board,positions,m,pieces,p)
        output <- AB_MINIMAX_R(upd$board,ply,-player,
                               upd$positions,upd$pieces,
                               depth+1,Alpha,Beta)

        if (output$value>v){
          v <- output$value
          best$value <- output$value
          best$pos <- positions[m,]
          best$piece <- pieces[p,]
          Alpha <- max(Alpha,v)
        }
        if (v>=Beta) return(best)
      }
    }
  }else{
    v <- Inf
    for (m in 1:nrow(positions)){
      for (p in 1:nrow(pieces)){
        upd <- Result_R(board,positions,m,pieces,p)
        output <- AB_MINIMAX_R(upd$board,ply,-player,
                               upd$positions,upd$pieces,
                               depth+1,Alpha,Beta)

```

```

        if (output$value<v){
          v <- output$value
          best$value <- output$value
          best$pos <- positions[m,]
          best$piece <- pieces[p,]
          Beta <- min(Beta,v)
        }
        if (v<=Alpha) return(best)
      }
    }
  }
}

return(best)
}

```

Podemos comprobar que funciona igual que la función original utilizando un ejemplo.

```

set.seed(41)
demo <- RAND.GAME_LIMITED(limit=12)

```

```

AB_MINIMAX(demo$board,ply=13,player=1,demo$positions_left,demo$pieces_left)

```

```

## $value
## [1] 9
##
## $pos
## [1] 1 4
##
## $piece
## [1] 1 3 5 7

```

```

AB_MINIMAX_R(demo$board,ply=13,player=1,demo$positions_left,demo$pieces_left)

```

```

## $value
## [1] 9
##
## $pos
## [1] 1 4
##
## $piece
## [1] 1 3 5 7

```

La diferencia en los tiempos de ejecución puede obtenerse utilizando el paquete microbenchmark.

```

microbenchmark::microbenchmark(
  'R'={AB_MINIMAX_R(demo$board,ply=13,player=1,demo$positions_left,demo$pieces_left)},
  'C++'={AB_MINIMAX(demo$board,ply=13,player=1,demo$positions_left,demo$pieces_left)}
)

```

```

## Unit: microseconds
## expr      min       lq      mean  median      uq      max neval  cld
##   R 13242.1 14604.3 20137.87 16809.8 22100.9 57384.4   100   b
##  C++  259.1   296.1   375.22   322.5   407.2  1144.8   100   a

```

## Referencias

Russell, S. y Norvig, P. (2020). Adversarial Search and Games. En S. Russell y P. Norvig, *Artificial Intelligence: a modern approach* (4a ed) pp.(146-180). Pearson.