Evidentemente, $nclears$ tendrá asociado un peso positivo, mientras que el resto de factores tendrán asociados pesos negativos, ya que representan penalizaciones a la hora de colocar una ficha.

Un inconveniente inmediato de esta primera aproximación es que no se contempla, de manera explícita, la construcción de **bloques consistentes** por parte del módulo de IA de la máquina. Es decir, sería necesario incluir la lógica necesaria para premiar el acoplamiento entre fichas de manera que se premiara de alguna forma la *colocación lógica* de las mismas. La figura 3.43 muestra un ejemplo gráfico de esta problemática, cuya solución resulta fundamental para dotar a la máquina de un comportamiento *inteligente*.

3.5.8. Caso de estudio. Tres en raya (tic-tac-toe) con NME

En esta sección se discuten los aspectos relacionados con el módulo de IA de un juego tres en raya o tic-tac-toe. Básicamente, la implementación de dicho módulo es una variación del **algoritmo minimax** discutido en la sección 3.5.6. Este juego de dos jugadores es un ejemplo sencillo, ya que es fácil averiguar que el juego perfecto siempre finaliza en empate, independientemente de quién comience, que permite ilustrar adecuadamente la implementación del módulo de IA.

La clase Board

Antes de discutir la implementación del algoritmo minimax es importante discutir la clase *board* o tablero, la cual almacena el **estado de juego** y permite reflejar la evolución del mismo. Esta clase, tal y como muestra el siguiente listado de código, contiene un array de elementos de tipo *TPlayer* (línea 13) que permite representar el estado de cada casilla. Dicho estado queda definido por el tipo enumerado *TPlayer* (línea 5), el cual alberga los valores *Empty* (casilla vacía), *Player_O* (jugador humano) y *Player_X* (máquina).

En esta solución se ha optado por utilizar un array unidimensional en lugar de uno bidimensional debido a que simplifica la implementación de la lógica del juego. De este modo, la longitud de este array será de 9, es decir, mantiene casillas desde la número 0 hasta la número 8.

Listado 3.57: Clase Board. Variables miembro y constructor.

```
1  // Enumeración para las opciones disponibles en una casilla.
2  // Empty ---->Casilla vacía.
3  // Player O -->Casilla con O.
4  // Player X -->Casilla con X.
5  enum TPlayer { Empty; Player_O; Player_X; }
6
7  class Board {
8
9    // Estado del tablero.
10   // - X O
11   // X X O
12   // - O -
13   private var _pieces:Array<TPlayer>;
14
15   public function new () {
16     _pieces = new Array<TPlayer>();
17     for (i in 0...9)
18       _pieces[i] = Empty;
19   }
20
21   // ...
22
23 }
```

De este modo, las acciones de **efectuar y deshacer movimientos** son triviales, como se expone a continuación.

Listado 3.58: Clase Board. Funciones makeMove() y undoMove().

```
1  // No comprueba si el movimiento es válido.
2  // move representa la casilla [0..8].
3  // player representa al jugador.
4  public function makeMove (move:Int, player:TPlayer) : Void {
5    _pieces[move] = player;
6  }
7
8  // Deshace un movimiento previamente realizado.
9  public function undoMove (move:Int) : Void {
10   _pieces[move] = Empty;
11 }
```

La clase *Board* también contiene la lógica necesaria para detectar si un jugador ha ganado. Para ello, la función *getWinner()* comprueba si alguna de las filas, columnas o diagonales contiene todas sus casillas del mismo tipo (siempre que no estén vacías).

Listado 3.59: Clase Board. Función getWinner().

```
1  // Devuelve, si existe, el ganador de una partida
2  // considerando el estado actual del tablero.
3  // Si no hay ganador, devuelve Empty.
4  public function getWinner () : TPlayer {
5    var winningStates:Array<Array<Int>> =
6      [
7        [0, 1, 2], [3, 4, 5], [6, 7, 8],
8        [0, 3, 6], [1, 4, 7], [2, 5, 8],
9        [0, 4, 8], [2, 4, 6],
10      ];
11
12   for (st in winningStates) {
13     var aux:Array<TPlayer> = [_pieces[st[0]],
14                   _pieces[st[1]],
15                   _pieces[st[2]]];
16     if (_pieces[st[0]] != Empty && allEqual(aux))
17       return _pieces[st[0]];
18   }
19   return Empty;
20 }
```

Por otra parte, otras dos funciones interesantes son las que permiten obtener la lista de movimientos válidos (función *getValidMoves()* en líneas 3-10) y comprobar si el juego ha terminado (función *isGameOver()* en líneas 15-17), respectivamente. La primera de ellas es trivial, ya que devuelve una lista con los números de las casillas que están vacías, es decir, sin ficha. La segunda comprueba dos casos: i) si no existen más movimientos válidos o ii) si ya hay un ganador.

Listado 3.60: Clase Board. Funciones getValidMoves() y isGameOver().

```
1  // Devuelve una lista que contiene las casillas
2  // con los movimientos posibles.
3  public function getValidMoves () : Array<Int> {
4    var validMoves:Array<Int> = new Array<Int>();
5    for (i in 0...9)
6      if (_pieces[i] == Empty)
7        validMoves.push(i);
8
9    return validMoves;
10 }
11
12 // Comprueba si el juego ha terminado...
13 // ¿Existe algún movimiento válido?
14 // ¿Hay un ganador?
15 public function isGameOver () : Bool {
16   return (getValidMoves().length == 0 || getWinner() != Empty);
17 }
```

La clase *Board* se utiliza tanto en la clase *Minimax*, la cual se discute a continuación, con en la clase *TicTacToe*, la cual es la clase principal

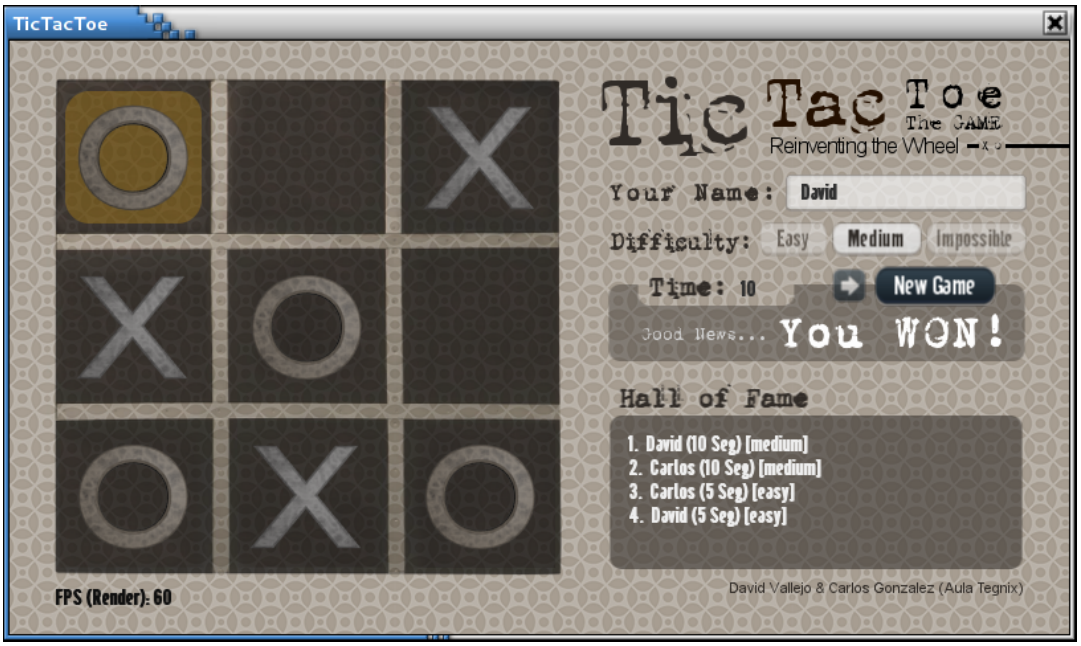

Figura 3.44: Captura de pantalla del juego Tic-Tac-Toe.

del juego y que se encarga de la gestión de toda la parte gráfica y de eventos.

La clase Minimax

La clase *Minimax* encapsula la funcionalidad asociada a una variante del algoritmo minimax, denominada negamax, que simplifica la lógica de dicho algoritmo. En particular, esta variante de búsqueda está basada en la propiedad *zero-sum*, la cual se basa en la siguiente propiedad:

$$max(a,b) = -min(-a,-b)$$

es decir, el valor de una posición para el jugador A es la negación del valor para el jugador B. De este modo, el jugador al que le toca mover busca un movimiento que maximice la negación del valor resultante de dicho movimiento. Esta posición sucesora debe ser, por definición, evaluada por el oponente. Este planteamiento garantiza la independencia con respecto al jugador que mueve, por lo que la implementación sirve de manera independiente al jugador al que le toca jugar en cada momento.

La simplificación del **algoritmo negamax** requiere que el jugador A seleccione el movimiento con el sucesor de máximo valor mientras que el jugador B elija el movimiento con el sucesor de menor valor.

El siguiente listado de código muestra la implementación de la función *buildTreeRec()*. Está función tiene un **planteamiento recursivo** y su principal responsabilidad es **construir el árbol de búsqueda** hasta una profundidad *depth* (pasada como parámetro) y devolver la mejor puntuación calculada.

Antes de pasar a discutir dicho código, es importante resaltar cómo se modela la **dificultad de la máquina** cuando un jugador se enfrenta a la misma. Actualmente, existen tres niveles de dificultad: *easy*, *medium* e *impossible*. Estos se modelan a través del nivel de recursión del algoritmo minimax. Actualmente, *easy* se modela con un nivel de recursión igual a 1, *medium* con un nivel de 3 y *impossible* con un nivel de 6. Como el lector podrá comprobar, no es posible vencer a la máquina en nivel de dificultad *impossible*.

En primer lugar, esta función contempla al principio el cambio de turno en las líneas 8–10 para que cuando, posteriormente, se realice una llamada recursiva (línea 31), entonces se llame con el jugador contrario. Así mismo, al principio se controlan los casos base de la recursividad, basados en detectar si existe algún ganador o si el juego quedó en empate (previamente se controla el nivel máximo de profundidad en la línea 23):

- El jugador actual gana y la función devuelve +INFINITO (línea 13).
- El oponente gana y la función devuelve -INFINITO (línea 14).
- Hay un empate y la función devuelve 0 (línea 15).

A continuación, la implementación planteada recupera la lista de posibles movimientos, denominada *movelist*, para el estado actual en la línea 18 y maneja una lista paralela, denominada *salist*, para almacenar la evaluación de cada uno de ellos, como se aprecia en la línea 22. En la variable *alpha* de la línea 20 se almacena la mejor puntuación y, por ese motivo, se inicializa a -INFINITO.

Como se ha mencionado anteriormente, el algoritmo minimax es un algoritmo recursivo. Precisamente, las siguientes líneas de código evalúan, de manera recursiva, las **opciones existentes** en la lista de posibles movimientos. El bucle for de las líneas 45–57 permite llevar a cabo tal tarea. La idea es sencilla, simular el movimiento en la línea 49 y cambiar el turno en la línea 51, efectuando una llamada recursiva y negando el signo del resultado obtenido, como ya se ha discutido en esta sección.

Listado 3.61: Clase Minimax. Función buildTreeRec().

```
1  // Construye recursivamente el árbol hasta una profundidad depth.
2  // Devuelve la mejor puntuación.
3  public function buildTreeRec (board:Board, currentPlayer:TPlayer,
4                    depth:Int) :Int {
5    // Para en profundidad maxDepth.
6    if (depth > _maxDepth) return 0;
7
8    // Para el cambio de turno...
9    var otherPlayer:TPlayer;
10   if (currentPlayer == Player_O) otherPlayer = Player_X;
11   else                           otherPlayer = Player_O;
12
13   var winner:TPlayer = board.getWinner();
14   if (winner == currentPlayer)   return INFINITY;
15   if (winner == otherPlayer)     return -INFINITY;
16   if (board.isGameOver())        return 0;
17
18   // Lista de posibles movimientos.
19   var movelist:Array<Int> = board.getValidMoves();
20   // Mejor puntuación.
21   var alpha:Int = -INFINITY;
22   // Lista con resultados (paralela a movelist)
23   var salist:Array<Int> = new Array<Int>();
24
25   // Obtiene los mejores resultados y asocia el mejor movimiento.
26   for (i in movelist) {
27     // Copia para evaluar el movimiento i.
28     var board_copy:Board = board.copy();
29     // Movimiento potencial.
30     board_copy.makeMove(i, currentPlayer);
31     // Cambio de turno (y de signo).
32     var subalpha:Int = -buildTreeRec(board_copy,
33                       otherPlayer,
34                       depth + 1);
35     if (alpha < subalpha) alpha = subalpha;
36
37     // Añade resultado después de la evaluación.
38     if (depth == 0) salist.push(subalpha);
39   }
40
41   // Si llega a profundidad 0 y se han explorado todos los
42   // sub-árboles, estudia la lista de mejores movimientos
43   // y se elige uno de ellos al azar para jugar.
44   if (depth == 0) {
45     var candidates:Array<Int> = new Array<Int>();
46     for (i in 0...salist.length)
47       if (salist[i] == alpha)
48     candidates.push(movelist[i]);
49     _bestMove = candidates[Std.random(candidates.length)];
50   }
51
52   // Devuelve la mejor puntuación.
53   // En bestMove se almacena la mejor opción.
54   return alpha;
55 }
```

Note el cambio de turno mediante la variable *otherPlayer* y la actualización del nivel de profundidad con la expresión *depth + 1*. A la vuelta de la recursividad se comparan los valores resultantes de evaluar los situaciones posibles, de manera que cada jugador se quede con la mejor (almacena su valor en *alpha*).

Finalmente, cuando se llega a profundidad 0 (líneas 62-68), habiendo estudiado los sub-árboles, se obtiene la lista con los **mejores movimientos** y se elige uno de ellos al azar. A continuación, se actualiza la variable miembro *_bestMove* (línea 67), para la siguiente jugada, y se devuelve la mejor puntuación (línea 72).

El siguiente listado muestra la llamada inicial a *buildTreeRec()*. Note cómo esta función devuelve el mejor movimiento de los realmente evaluados (línea 5).

Listado 3.62: Clase Minimax. Función buildTree().

```
1 public function buildTree (board:Board, currentPlayer:TPlayer) : Int
      {
2   _bestMove = -1;
3   // Llamada a la función recursiva.
4   var alpha:Int = buildTreeRec(board, currentPlayer, 0);
5   return _bestMove;
6 }
```

La llamada desde el **código de Tic-Tac-Toe** se realiza a través de la función *makeMovement()* de la clase *TicTacToe*. La generación del árbol de búsqueda se realiza a través de la llamada *buildTree()* en la línea 2. Ésta llamada sólo se realiza a la hora de efectuar el movimiento inicial (ver valor por defecto del parámetro *move* en la línea 1).

Listado 3.63: Clase TicTacToe. Función makeMovement().

```
1 function makeMovement(player:TPlayer, move:Int=-9):Void {
2   if (move == -9) move = _miniMax.buildTree(_board, player);
3   var moves:Array<Int> = _board.getValidMoves();
4   if (Lambda.has(moves, move)) {
5     _board.makeMove(move, player);
6     updateTab(move, player);
7     changeTurn();
8   }
9 }
```