Fundamentos de Inteligencia Artificial

Práctica 1

El juego de las damas simplificado

Autor	Cristian Aguilera Martínez
Turno	jueves 11:30 – 13:00
Profesor	Fidel Aznar Gregori
Fecha de entrega	25 de octubre de 2009

Índice

1. Introducción a los juegos de mesa

En los juegos de mesa o en los juegos por turnos, como el ajedrez o las damas, es fácil desarrollar un sistema de inteligencia artificial que pueda jugar realizando movimientos como si fuera un jugador humano, cambiando la posición de las fichas que hay sobre el tablero. Las reglas de estos juegos son sencillas, lo que permite dedicar más tiempo a la parte de inteligencia artificial y menos tiempo al resto del código. Suelen ser juegos fáciles de mostrar por pantalla, sin conocimientos especializados en motores 3D, librerías gráficas o interfaces de usuario.

Las técnicas de IA aplicadas a estos juegos se basan en la búsqueda de la mejor jugada posible para vencer la partida partiendo de un estado del juego concreto. Este estado del juego puede ser una partida recién comenzada, con las fichas del tablero en su posición inicial, o una partida sobre la que ya se han efectuado una serie de movimientos sobre el tablero.

Si dado un estado del juego –un tablero y unas fichas, en el caso de las damas– somos capaces de generar todas las posibles combinaciones de movimientos que se pueden hacer sobre él y obtener una serie de posibles resultados finales de la partida que se está jugando, podríamos elegir el movimiento que nos permita alcanzar, en futuros turnos de la partida, el mejor de todos los resultados generados. Al hacerlo, estamos guiando al rival hacia un resultado de la partida que nos favorece.

Sin embargo, es inviable generar todas las combinaciones de la partida ya que nos encontramos con tantas combinaciones como la cantidad de movimientos permitidos en una tirada elevado a la cantidad de tiradas necesarias para finalizar la partida. Por ejemplo, en el ajedrez habría que evaluar 35^{100} situaciones[1] para poder elegir la mejor. Esa cantidad de estados presenta un problema en el tiempo y en el espacio; se necesitaría mucha memoria que almacenara todos los estados y se necesitaría mucho tiempo para decidir qué movimiento realizar. Aunque en las damas la cantidad de estados sería inferior, también es inabordable.

Existen técnicas de IA –algoritmos– que realizan una aproximación a esta búsqueda, permitiendo encontrar movimientos que nos lleven a un estado aceptable, aunque no se garantiza que dicho estado sea el mejor de todos. Los algoritmos que estudiaremos serán Minimax, Alfa-beta y SSS*.

El algoritmo Minimax examina todas las soluciones pero sólo hasta una cantidad de tiradas determinada, generando un árbol de estados de un tamaño finito y abordable. El algoritmo Alfa-beta

1 35 es la cantidad media de movimientos posibles en una jugada; 100 es el número medio de jugadas que son necesarias para finalizar una partida.

presenta una mejora respecto a Minimax que se basa en la poda –descarte– de las ramas de dicho árbol que no son potencialmente aceptables. Por último, el algoritmo SSS* realiza la misma función que Minimax y Alfa-beta, pero consiste en el mantenimiento de una lista ordenada con los posibles estados de la partida y no en un árbol. Cuando Alfa-beta realiza una poda, SSS* también la realiza, pero no al revés; SSS* puede podar una rama que Alfa-beta examinaría.

Estos algoritmos pueden utilizarse en juegos basados en turnos con dos o más jugadores y que no contemplen la posibilidad de azar. Es necesario que, dado un tablero y una posición de las fichas de la partida del juego, se pueda generar un abanico de estados que serían el resultado de aplicar todos los movimientos posibles de cada ficha sobre el tablero. Además, cada uno de esos tableros tendrá un valor asociado[2] que permitirá ordenarlos de mayor a menor para elegir el mejor después.

2. El juego de las damas simplificado

En esta práctica se ha pedido se ha pedido implementar el juego de las damas pero con unas reglas reducidas que no se corresponden con el juego real. La idea es realizar un juego de las damas más simple. Para ello, se ha proporcionado una plantilla en Java en la que sólo falta implementar un algoritmo de búsqueda para decidir qué movimiento debe realizar la IA en una situación concreta del tablero.

Reglas de este juego:

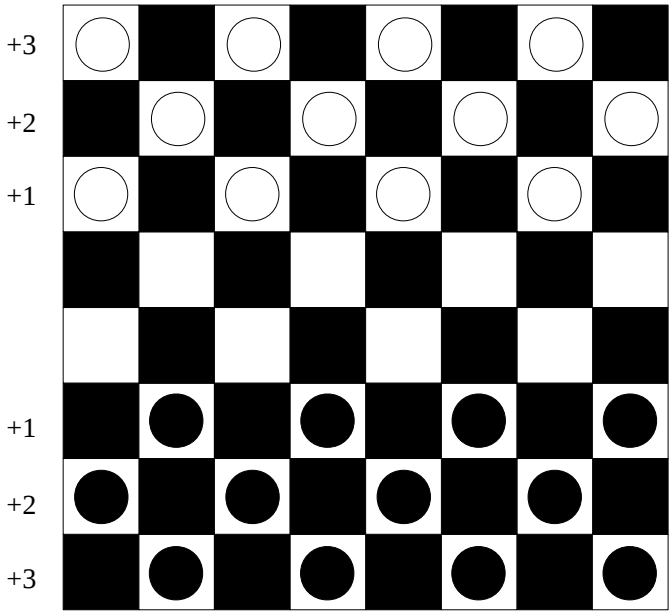
- Cada ficha tiene 4 movimientos posibles: hacia la derecha, hacia la derecha comiendo, hacia la izquierda y hacia la izquierda comiendo.
- No existe el movimiento múltiple; sólo se puede comer una ficha en cada turno.
- Cuando una ficha llega al extremo del tablero del jugador rival, no puede retroceder. Permanecerá en esa posición hasta el final de la partida.
- La partida finaliza cuando alguno de los dos jugadores no puede mover ninguna ficha.

Una vez finalizada la partida, se debe calcular la puntuación de cada jugador y vence aquél que tenga una puntuación mayor:

- Cada ficha del tablero suma 1 punto al jugador al que pertenece.
- Cuanto más cerca estén las fichas del jugador en el extremo del tablero del jugador rival, más puntuación recibe esa ficha:

2 Este valor lo calcularemos con una función de evaluación.

Extremo del jugador negro



Extremo del jugador blanco

Los posibles modos de juego son: *humano contra humano*, *humano contra máquina* y *máquina contra máquina*. Sólo en el primero de los modos no es necesario calcular el movimiento de la IA, porque no existe.

2.1. Función de evaluación

Como ya se ha dicho anteriormente, para el juego de las damas es necesaria una función que nos permita evaluar un tablero. Esta función nos dirá cuánto de bueno es un tablero dada la posición de las fichas. Por ejemplo, una posible evaluación sería contabilizar la cantidad de fichas que tengo y la cantidad de fichas que tiene mi rival, devolviendo la diferencia entre ambas cantidades. Otra posible función de evaluación podría contemplar sólo que las fichas lleguen al extremo del tablero del rival, dando una mejor evaluación cuando haya más fichas en el otro extremo.

Supongamos que la función de evaluación sólo calculará el número de fichas. Esta función de evaluación nunca comprobará que el tablero sea bueno en futuras tiradas. Es decir, si en el tablero actual tengo más fichas que mi rival, pero en el siguiente turno mi rival hace un movimiento que mata una de mis fichas y gana la partida, la función de evaluación dirá que el tablero es bueno igualmente. Aunque sea inevitable que vaya a perder la partida en el siguiente movimiento, la función de evaluación dirá que es un buen tablero porque nunca explorará en los niveles inferiores. Si lo hiciera, el coste computacional de la función sería muy elevado.

El hecho de no poder explorar todas las posibles combinaciones de tableros, nos obliga a usar una función de evaluación que, a priori, nos diga qué tablero es potencialmente mejor que el resto. Esta función debe realizar pocos cálculos y en el menor tiempo posible. Si pudiéramos explorar todos los tableros, la mejor función de evaluación sería aquella que se corresponde con la función que calcula la puntuación de la partida.

Para el desarrollo de la práctica se han utilizado distintas funciones de evaluación y que en los apartados siguientes analizaremos para saber cuál es la mejor función de evaluación, que será aquella que nos permita ganar más partidas.

Función de evaluación 1	Puntuación que tendría el jugador 1 en el tablero restándole la que tendría el jugador 2 en el tablero.
Función de evaluación 2	Sólo se cuenta la cantidad de fichas de las blancas y se resta la cantidad de fichas de las negras.
Función de evaluación 3	Similar a la función de evaluación 1, pero puntuando también que las fichas estén en un extremo del tablero, de manera que no puedan morir, al menos, en el siguiente movimiento.
Función de evaluación 4	Similar a la función de evaluación 1, pero puntuando también que las fichas estén en la parte central del tablero, dando un mayor control sobre el juego.

En estas funciones de evaluación, siempre se obtiene el valor del tablero considerando que las negras son el rival y que las blancas son el jugador que quiere realizar el movimiento. De esta manera, las blancas siempre intentarán elegir el tablero con una función de evaluación más alta y las negras intentarán minimizar dicho valor.

En el fichero *Tablero.java* se encuentran los métodos de la clase *Tablero* que se corresponden con estas funciones de evaluación.

2.2. Generación de estados

En apartado 1 ya se dijo que era necesario generar todos las posibles situaciones diferentes partiendo de un estado o tablero inicial.

En un tablero determinado, cada jugador puede realizar una serie de movimientos. Por cada uno de esos movimientos, se puede construir otro tablero como resultado de efectuar ese movimiento en el tablero original. De esta manera, la cantidad de tableros que obtendremos coincide con la cantidad de movimientos que pueden efectuarse en el tablero actual. Habrá que tener en cuenta que sólo se generarán los tableros correspondientes a los movimientos de uno de los jugadores, que será el jugador que esté buscando un movimiento a realizar.

Primero hay que obtener una lista con los movimientos que puede ejecutar el jugador. Es necesario recorrer el tablero, comprobando qué fichas pertenecen al jugador. Según el juego simplificado de las damas, cada una de esas fichas tiene 4 movimientos posibles: hacia a la izquierda o hacia la derecha, comiendo o sin comer. Sin embargo, como mucho podrá elegir entre 2 movimientos, ya que si puede ir hacia a la izquierda comiendo, no podrá ir hacia la izquierda sin comer; es decir, como máximo 2 movimientos posibles a la vez. En un tablero, habrá un máximo de 24 movimientos posibles.[3]

En una partida *máquina contra máquina*, el resultado de la partida debe ser idéntico independientemente de que comiencen a jugar las blancas o las negras; sólo debe cambiar el color del vencedor. Por ejemplo, si comienzan blancas y el resultado es 10-3, cuando comiencen negras el resultado debe ser 3-10. Lo que se pretende es que las partidas sean simétricas. Por eso, cuando se recorre el tablero, se hace en distinto orden según el color del jugador. Además, en las blancas se da prioridad al movimiento hacia a la izquierda y a las negras se le da prioridad al movimiento hacia la derecha.

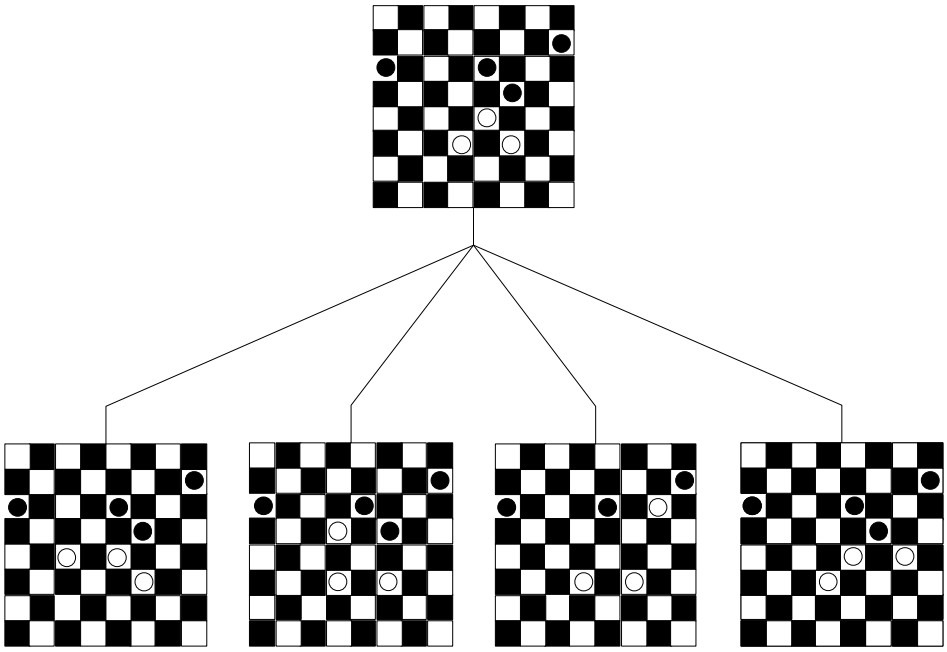
Una vez se tiene la lista de movimientos, se realizan copias del tablero original; tantas como movimientos haya en esa lista. Para cada copia del tablero, se extraerá un movimiento de la lista y se efectuará, generando un tablero único. La cantidad de tableros generados es lo que se llama el factor de ramificación del árbol que se generará al ejecutar alguno de los algoritmos. Es el número de hijos que se pueden extraerse desde el tablero original.

En la práctica, se ha incluido un método en la clase *Tablero*, con el nombre de *movimientosPo-*

3 En un tablero puede haber hasta 12 fichas de un mismo color. Si cada ficha tiene entre 0 y 2 movimientos, sólo tendremos 12·2 = 24 movimientos máximos posibles.

sibles, que devuelve una lista con los posibles movimientos de un jugador, siendo el jugador aquél que se indique como parámetro en ese método. Para generar la copia de los tableros, se hace eso del constructor de copia de la clase *Tablero*, que duplica un tablero en un nuevo espacio de memoria. Cuando la lista de movimientos ha sido obtenida, se recorre un bucle que duplica el tablero original y aplica un movimiento de la lista en dicho tablero utilizando el método *hacerTirada* también de la clase *Tablero*.

Ejemplo de cómo se generarían los tableros hijos de un tablero:



3. Algoritmos de búsqueda implementados

Como ya se ha dicho, en esta práctica se implementará el Minimax, el Alfa-beta y el SSS*. Para ello, haremos uso de alguna de las funciones de evaluación diseñadas y de la función generadora de nodos que se ha estudiado en el apartado anterior.

3.1. Algoritmo Minimax

Éste algoritmo es el más simple y también es el más lento. Se ha implementado como fase previa de la implementación del algoritmo Alfa-beta que ayude a depurarlo.

```
Algoritmo Minimax: V(N)
Entrada: Nodo N
Salida: Valor minimax de dicho nodo

Si N es nodo hoja entonces devuelve f(N)
sino
   Generar los nodos hijos de N: N1, N2, …, Nb (b: número de hijos)
   Calcular el valor minimax de los nodos hijos: V(N1), V(N2), …, V(Nb)
   Si N es un nodo MAX entonces devuelve max(V(N1), …, V(Nb))
   sino devuelve min(V(N1), …, V(Nb))
   FinSi
FinSi
```

Cuando un jugador máquina tenga que elegir un movimiento entre todos los posibles, ejecutará el algoritmo pasándole como parámetro el nodo con el tablero actual.

Según el pseudocódigo, ante un nodo hoja, hay que devolver su valor asociado, que se calcula con una de las funciones de evaluación estudiadas anteriormente.

Un nodo hoja es aquél que no tiene hijos. Esto puede suceder de dos maneras: que en el tablero de ese nodo no se pueda realizar ningún movimiento o que ya se haya alcanzado el número de niveles máximos en el árbol, lo que obliga a llevar la cuenta del nivel actual dentro de cada nodo.

Los nodos hijos se generan tal como se indicó en el apartado anterior. Después hay que aplicar la recursividad a cada uno de los hijos para decidir con cuál quedarnos: con el mayor si es un nodo MAX o con el menor si es un nodo MIN.

Hay varias formas de decidir si un nodo es MAX o MIN y que no son necesariamente equivalentes. Normalmente, un nodo es MAX si el jugador asociado al nodo se corresponde con el jugador que finalmente realizará la tirada sobre el tablero. En el árbol generado, los nodos de los niveles impares serían nodos MAX y el resto serían nodos MIN. En la práctica no se ha hecho así, sino que se ha determinado que un nodo es MAX si el jugador asociado son las blancas y MIN si son las negras; siempre es así. De esta forma, el primer nivel del árbol ya no es necesariamente un nodo MAX, sino que puede comenzar siendo un nodo MIN cuando sean las negras las que han iniciado el algoritmo Minimax. Se puede hacer así porque la función de evaluación devuelve el valor del tablero independientemente de qué jugador tenga que mover. Así, los nodos MAX (blancas) siempre intentarán maximizar el valor *minimax* y los nodos MIN (negras) siempre intentarán minimizarlo.

En la práctica, el algoritmo Minimax se implementa en el método *minimax* de la clase *JugadorMaquina*. Los parámetros que recibe son:

- *Tablero tablero*: Es el tablero sobre el que se va a trabajar.
- *int jugador*: Número del jugador. 1 si es blancas o 2 si es negras. Sirve para clasificar el nodo como MAX o MIN.
- *int nivelActual*: Número del nivel actual. Se decrementa en cada llamada recursiva.
- *int nivelesTotales*: Cantidad de niveles máximos que explorará el algoritmo.

3.2. Algoritmo Alfa-beta

Respecto a Minimax, el algoritmo Alfa-beta sólo presenta una mejora de eficiencia. Añade una función de poda que reduce los cálculos y el tiempo de ejecución.

```
Algoritmo Alfa-beta: V(N, α, β)
Entrada: Nodo N, valores α y β
Salida: valor minimax de dicho nodo

Si N es nodo hoja entonces devuelve f(N)
sino
   Generar los nodos hijos de N: N1, N2, …, Nb (b: número de hijos)
   Si N es un nodo MAX entonces
      Para k = 1 hasta β hacer
         α = max[α, V(Nk, α, b)]
         Si α ≥ β entonces devolver β FinSi
         Si k = b entonces devolver α FinSi
      FinPara
   sino
      Para k = 1 hasta β hacer
         β = min[β, V(Nk, α, β)]
         Si α ≥ β entonces devolver α FinSi
         Si k = b entonces devolver β FinSi
      FinPara
   FinSi
FinSi
```

En este caso, cuando el jugador máquina quiera escoger entre uno de los movimientos posibles, ejecutará el algoritmo pasándole como referencia el nodo con el tablero y el jugador, blanco o negro, asociado. Los valores iniciales de α y β serán -∞ y +∞ respectivamente.

De momento, tanto el algoritmo Alfa-beta como Minimax, sólo nos devuelve el valor *minimax* del mejor nodo al que podemos acceder, pero eso no es lo que necesitamos. Cuando se ejecuta alguno de estos dos algoritmos, lo que se busca es el mejor movimiento que se puede efectuar, el que nos permita alcanzar la mejor situación final posible; con el valor *minimax* no vamos a ningún lado. Es necesario que el algoritmo nos indique qué movimiento tenemos que elegir para alcanzar el nodo de ese valor *minimax* que sí nos devuelve.

Una posible solución a este problema es añadir una variable global que se actualice cada vez que el valor de α o β cambien dentro del bucle *for*. Además, al ser un algoritmo recursivo, hay que tener en cuenta que sólo debe actualizarse si el cambio se produjo desde el nivel 2 del árbol.

En la práctica, el algoritmo Alfa-beta se implementa en el método a*lfaBeta* de la clase *JugadorMaquina*. Recibe los mismos parámetros que la implementación de Minimax añadiendo 2 más:

- *Tablero tablero*: Es el tablero sobre el que se va a trabajar.
- *int jugador*: Número del jugador. 1 si es blancas o 2 si es negras. Sirve para clasificar el nodo como MAX o MIN.
- *int nivelActual*: Número del nivel actual. Se decrementa en cada llamada recursiva.
- *int nivelesTotales*: Cantidad de niveles máximos que explorará el algoritmo.
- *int alpha*: Valor α que se propaga en cada llamada recursiva.
- *int beta*: Valor β que se propaga en cada llamada recursiva.

3.3. Algoritmo SSS*

Este algoritmo tiene la misma función que los dos anteriores. Devuelve el valor *minimax* y pretende encontrar el mejor movimiento posible. La principal diferencia con los otros algoritmos es que no es recursivo –y no genera un árbol de llamadas recursivas–, sino que ejecuta un proceso iterativo dentro de un bucle. Utiliza una lista ordenada para almacenar los sucesivos nodos que se generan en cada iteración del algoritmo.

```
Algoritmo SSS*: V(Ni)
Entrada: Nodo Ni
```

```
Salida: Valor minimax de dicho nodo

Insertar en LISTA el estado inicial N_i con N_i.s = VIVO y N_i.h = +∞
Hacer siempre
   Seleccionar y eliminar el primer estado de la lista: N
   Si N es Ni y N.s = SOLUCIONADO entonces devolver N.h FinSi
   Si N.s = VIVO entonces
      Si N no es terminal entonces
         Si N es MAX entonces
            Generar los nodos hijos de N: N_1, N_2, …, N_b (b: núm. de hijos)
            Para n = b hasta 1 hacer
               Insertar el estado N_n con N_n.s = N.s y N_n.h = N.h en
               la cabeza de LISTA.
            FinPara
         sino
            Generar el primer nodo hijo de N: N1
            Insertar el estado N1 con N1.s = N.s y N1.h = N.h en la
            cabeza de LISTA.
         FinSi
      sino
         Insertar el estado N con N.s = SOLUCIONADO y
         N.h = min(N.h, f(N)) delante de todos los estados con menor h
         que este estado.
      FinSi
   sino
      Sea N ≡ M_n (es decir, N es el hijo número n de otro nodo M)
      Si N es MAX entonces
         Si n ≠ b entonces
            Insertar M_n+1 con M_n+1.s = VIVO y M_n+1.h = N.h en la cabeza
            de LISTA
         sino
            Insertar M con M.s = SOLUCIONADO y M.h = N.h en la
            cabeza de LISTA.
         FinSi
      sino
         Insertar M con M.s = SOLUCIONADO y M.h = N.h en la
         cabeza de LISTA.
         Eliminar de LISTA toda la descendencia de M.
      FinSi
   FinSi
```

```
FinHacer
```

Los nodos en este algoritmo se dividen en dos estados: vivos o solucionados. Cuando un nodo haya sido explorado y toda su descendencia haya sido explorada, el nodo se convierte en un nodo solucionado. Cuando se trata de un nodo terminal, pasa directamente al estado solucionado, ya que no tiene descendencia –un nodo es terminal si su tablero asociado no tiene movimientos posibles o se ha alcanzado el nivel máximo de expansión del árbol–.

En cada iteración del algoritmo se extrae un nodo –nodo actual– de la lista para procesarlo según una serie de criterios. Estos criterios se pueden clasificar en 6 casos distintos, y cada nodo que se extrae de la lista entrará en alguno de estos casos:

Caso 1 *VIVO, no terminal, MAX*	Se introducen todos su hijos en la cabeza de la lista, en orden inverso, y propagando el estado y el valor del nodo actual.
Caso 2 *VIVO, no terminal, MIN*	Se introduce el primer hijo en la cabeza de la lista, propagando el estado y el valor del nodo actual.
Caso 3 *VIVO, terminal*	Se vuelve a introducir el nodo en la lista, cambiando su estado a solucionado y con un valor igual al mínimo entre el valor actual y el valor que devuelve la función de evaluación del tablero asociado al nodo.
Caso 4 *SOLUCIONADO, MAX, no es el hijo menor*	Se inserta el hermano de la derecha, con el estado vivo y con el valor del nodo actual.
Caso 5 *SOLUCIONADO, MAX, es el hijo menor*	Se inserta el padre del nodo actual, con el estado solucionado y con el valor del nodo actual.

Caso 6 *SOLUCIONADO, MIN*	Se inserta el padre del nodo actual, con el estado solucionado y con el valor del nodo actual. Se elimina toda la descendencia de este padre.

Este bucle continúa hasta que se extraiga el nodo raíz y ya esté solucionado.

En la práctica, se ha construido una clase *Nodo* dentro de *JugadorMaquina*. Este nodo contiene un tablero, un estado s y un valor h, además de otros valores auxiliares utilizados en la implementación del algoritmo. El método *SSS* de la clase *JugadorMaquina*, que es la implementación de SSS*, utiliza este tipo de objeto *Nodo*. Los parámetros que recibe la implementación de SSS* son:

- *Tablero tablero*: Es el tablero sobre el que se va a trabajar.
- *int jugador*: Número del jugador. 1 si es blancas o 2 si es negras. Sirve para clasificar el nodo como MAX o MIN.

4. Fase de experimentación

4.1. Análisis de juego

En apartados anteriores se propusieron distintas funciones de evaluación. Con el fin de comprobar cuál es mejor, se van a realizar una serie de partidas en modo *máquina contra máquina* enfrentando entre sí las distintas funciones de evaluación.

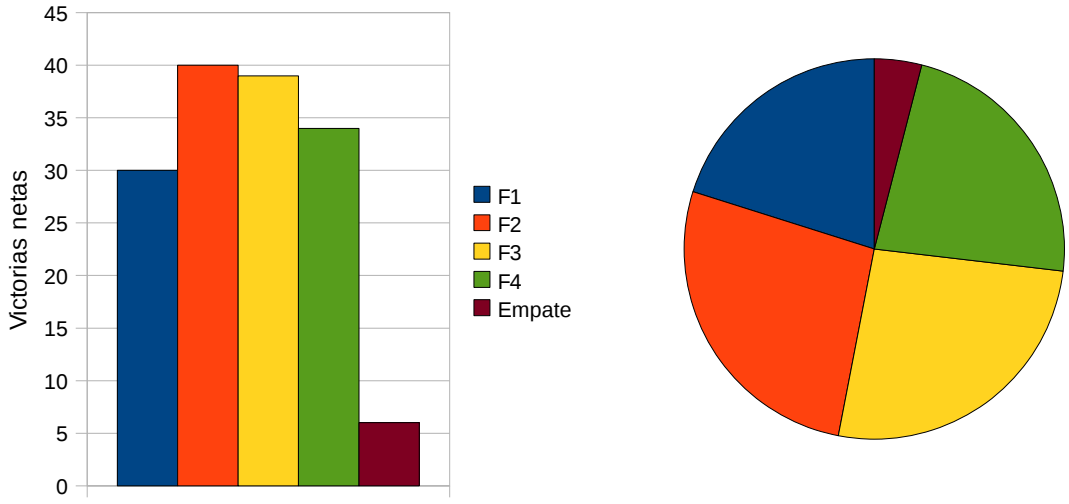
Para cada par de funciones de evaluación, se jugarán partidas en todos los niveles posibles que puedan procesarse con el algoritmo Alfa-beta –desde el 1 hasta el 10–. También se jugarán 10 partidas con la particularidad de que, en cada turno, se seleccionará aleatoriamente un nivel distinto. Con esta componente de azar, concertamos más partidas que nos permiten alcanzar una conclusión determinante sobre qué función de evaluación es mejor.

	Victorias del equipo local	Empates	Derrotas del equipo visitante
F1 contra F2	10	1	14
F1 contra F3	9	1	15

F1 contra F4	11	0	14
F2 contra F3	12	2	11
F2 contra F4	14	0	11
F3 contra F4	14	2	9

En total, se han realizado 150 combates, con el siguiente índice de victorias por función de evaluación:

	F1	F2	F3	F4	Empate
Victorias (%)	20	26,7	26	22,7	4
Victorias netas	30	40	39	34	6



Según los resultados, podemos decir que la función de evaluación 2 es ligeramente superior al resto de funciones.

4.2. Garantizando el correcto funcionamiento

Para comprobar que los algoritmos se han implementado correctamente, se han realizado una serie de pruebas. Estas pruebas consisten en generar gran cantidad de tableros aleatorios, calculando el valor *minimax* con cada uno de los 3 algoritmos y comprobando que esos valores coinciden siempre. Si los valores coinciden, podremos decir que los 3 algoritmos están correctamente implementados –o que todos tienen el mismo error, lo que resulta menos probable–.

Concretamente, se han calculado tantos valores *minimax* como indica la tabla:

	Nivel 1	Nivel 2	Nivel 3	Nivel 4	Nivel 5	Nivel 6	Nivel 7
Minimax	100.000	100.000	100.000	10.000	1.000		
Alfa-beta						1.000	500
SSS*							

A partir del nivel 5, el algoritmo Minimax consume mucho tiempo de cálculo y se excluye de las pruebas. Por otro lado, las pruebas sólo llegan hasta el nivel 7, porque SSS* tampoco es capaz de alcanzar el nivel 8. Además, conforme se va incrementando de nivel, se va reduciendo el número de tableros aleatorios que se generan.

Aunque el resultado de las pruebas sea satisfactorio, sólo llegamos a la conclusión de que, *probablemente*, no hay ningún error en la implementación.

4.3. Evaluación de rendimiento

4.3.1. Comparativa de los 3 algoritmos

A continuación se muestra una tabla con el tiempo resultante de ejecutar los 3 algoritmos con una serie de tableros aleatorios en los 7 primeros niveles.

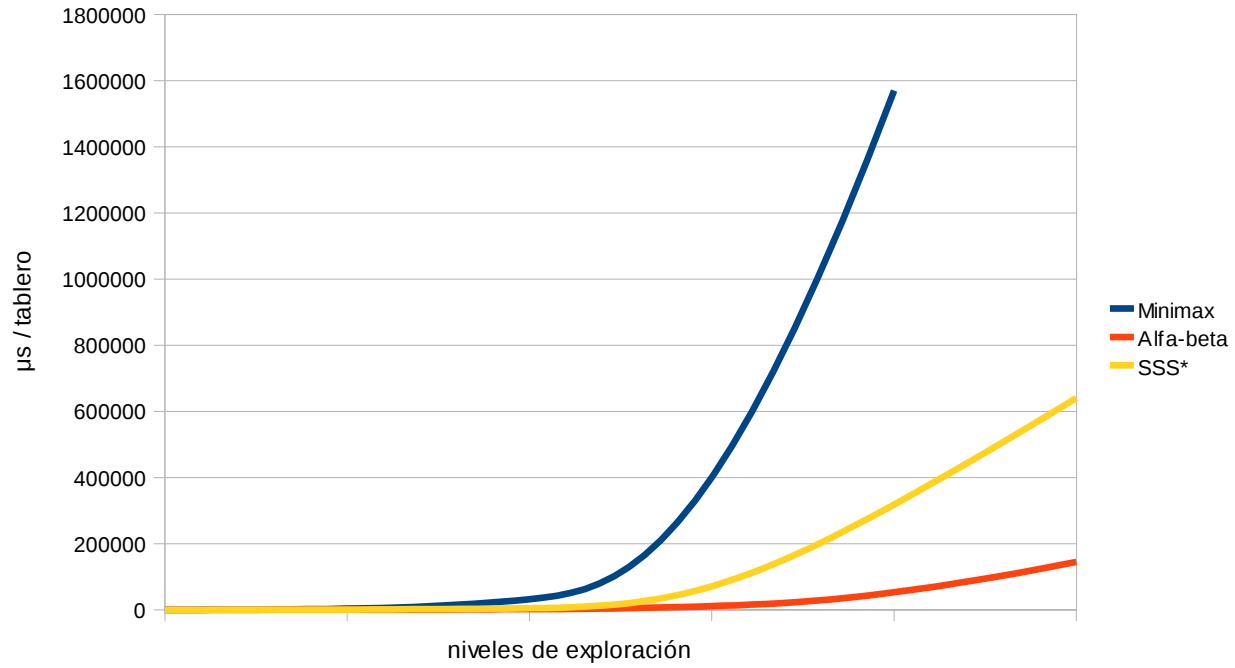
	Minimax	Alfa-beta	SSS*
100.000 tableros en nivel 1	0,527 s	0,392 s	0,976 s
100.000 tableros en nivel 2	4,947 s	4,595 s	9,705 s
100.000 tableros en nivel 3	57,226 s	21,2155 s	40,987 s
10.000 tableros en nivel 4	81,699 s	14,867 s	34,448 s
1.000 tableros en nivel 5	112,42 s	6,261 s	13,898 s
1.000 tableros en nivel 6	-	36,025 s	306,665 s
10 tableros en nivel 6	15,701 s	-	-
500 tableros en nivel 7	-	72,685 s	319,956 s
10 tableros en nivel 7	270,96 s	-	-

Para el algoritmo Minimax se han realizado 2 pruebas distintas con menos tableros ya que los tiempos eran muy largos.

Si normalizamos la tabla anterior, obtenemos una relación del tiempo medio aproximado que es necesario para calcular el movimiento de un único tablero en los distintos niveles que se han evaluado en esta prueba de rendimiento.

	Minimax	Alfa-beta	SSS*
µs / tablero en nivel 1	5,27	3,92	9,76
µs / tablero en nivel 2	49,47	45,95	97,05
µs / tablero en nivel 3	572,26	212,155	409,87
µs / tablero en nivel 4	8169,9	1486,7	3444,8
µs / tablero en nivel 5	112420	6261	13898
µs / tablero en nivel 6	1570100	36025	306665
µs / tablero en nivel 7	27096000	145370	639912

La siguiente gráfica muestra cómo evolucionan los algoritmos en los distintos niveles. Se puede prever la forma que tendrá la curva en los siguientes niveles.



La ejecución de Minimax crece rápidamente y es inviable para una cantidad de niveles superior a 7. Minimax nunca se lleva a la práctica y sólo es utilizado como un paso previo para la implemen-

tación de Alfa-beta.

Por otro lado, aunque Alfa-beta es recursivo, está dando mejores tiempos que SSS*. Para esta implementación, que calcula un movimiento para el juego de las damas, no compensa mantener una lista ordenada. Como veremos en los apartados siguientes, SSS* invierte el mayor tiempo de ejecución en ordenar la lista y eliminar la descendencia de los nodos. En cualquier caso, esto es sólo una explicación posible de por qué es más lento que Alfa-beta; por ejemplo, podría existir un error en la implementación que redujera drásticamente la eficiencia del algoritmo –aunque ningún compañero ha conseguido que su implementación sea más rápida que Alfa-beta–.

El siguiente fragmento ha sido extraído de la Wikipedia en inglés:

> *Stockman speculated that SSS* may therefore be a better general algorithm than alpha-beta. However, Steve Rozen and Judea Pearl have shown that the savings in the number of positions that SSS* evaluates relative to alpha/beta is limited and generally not enough to compensate for the increase in other resources (e.g., the storing and sorting of a list of nodes made necessary by the best-first nature of the algorithm).*

4.3.2. Estadísticas interesantes del algoritmo Alfa-beta

En este apartado estudiaremos algunas propiedades del algoritmo Alfa-beta.

La siguiente tabla muestra el número medio de llamadas recursivas que se lanzan en los distintos niveles al calcular el movimiento para una serie de tableros. También presenta la cantidad de podas que se han realizado:

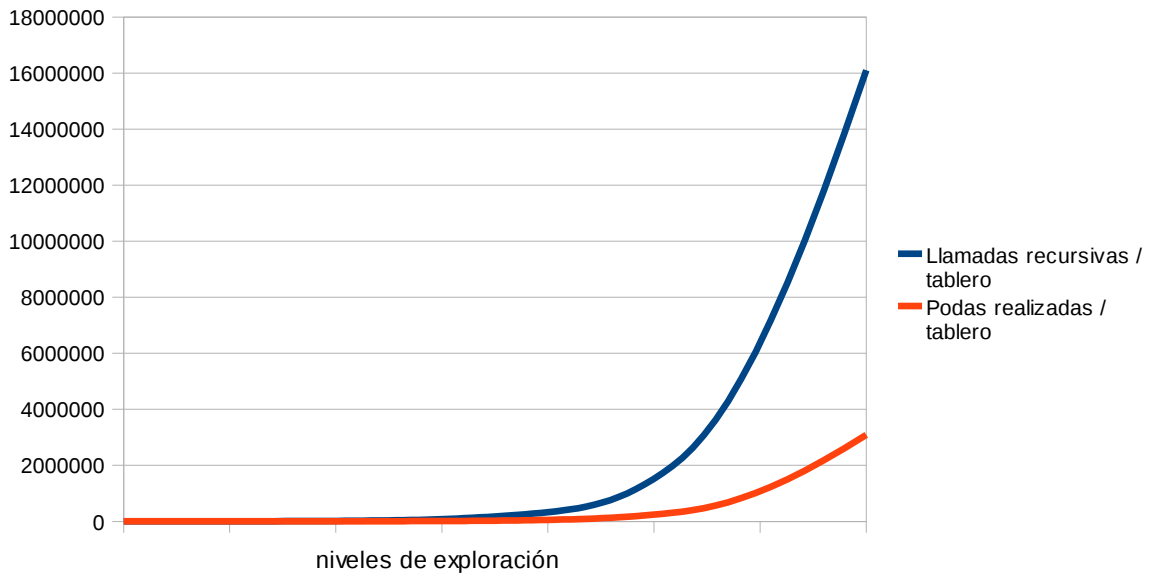
	Llamadas recursivas	Podas realizadas
100.000 tableros en nivel 1	100.000	0
100.000 tableros en nivel 2	1.466.810	0
100.000 tableros en nivel 3	7.265.763	1.169.591
100.000 tableros en nivel 4	50.609.487	4.972.673
100.000 tableros en nivel 5	210.164.968	38.405.991
10.000 tableros en nivel 6	123.470.557	15.725.776
1.000 tableros en nivel 7	48.995.812	9.178.815
500 tableros en nivel 8	137.698.184	18.938.053

100 tableros en nivel 9	81.913.249	16.192.449
100 tableros en nivel 10	480.510.246	67.655.284
10 tableros en nivel 11	161.025.763	30.853.922

Normalizando la tabla, obtenemos:

	Llamadas recursivas / tablero	Podas realizadas / tablero
Nivel 1	1	0
Nivel 2	14,67	0
Nivel 3	72,66	11,7
Nivel 4	506,09	49,73
Nivel 5	2101,65	384,06
Nivel 6	12347,06	1572,58
Nivel 7	48995,81	9178,82
Nivel 8	275396,37	37876,1
Nivel 9	819132,49	161924,49
Nivel 10	4805102,46	676552,84
Nivel 11	16102576,3	3085392,2

Su representación gráfica es la siguiente:



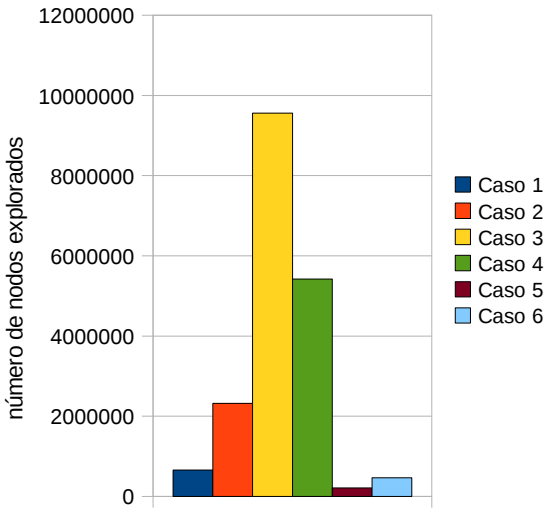
4.3.3. Estadísticas interesantes del algoritmo SSS*

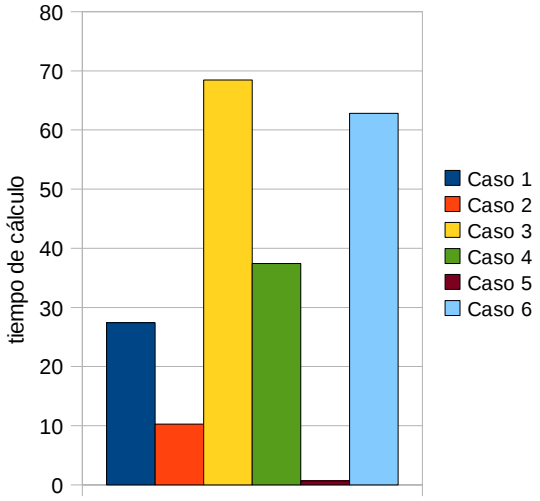
Como ya vimos en apartados anteriores, SSS* se compone de 6 casos en los que puede clasificarse cada uno de los nodos que se procesa. En la siguiente prueba evaluaremos qué cantidad tiempo dedica SSS* a cada uno de esos casos.

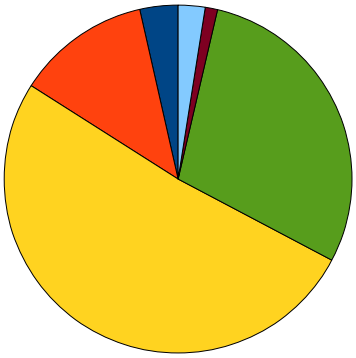
Se va a ejecutar el algoritmo 2100 veces –300 tableros aleatorios en cada uno de los 7 primeros niveles–. Las siguientes tablas muestran los resultados:

Total de nodos explorados	18644840
Tiempo total de ejecución	3m 27s

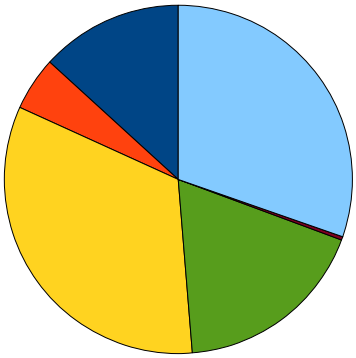
	Caso 1	Caso 2	Caso 3	Caso 4	Caso 5	Caso 6
Nº de nodos	658325	2323172	9561272	5422814	212122	467133
Nº de nodos (%)	3,53	12,46	51,28	29,08	1,14	2,51
Tiempo de cálculo (s)	27,428	10,279	68,438	37,396	0,645	62,81
Tiempo de cálculo (%)	13,25	4,97	33,06	18,07	0,31	30,34







Porcentaje de nodos explorados por cada caso



Porcentaje de tiempo de cálculo de cada caso

En las gráficas podemos ver que el caso 3 y el caso 6 son los que mayor coste temporal presentan. Aunque el caso 6 sólo procesa el 2,51% de los nodos, en él se invierte alrededor de un 30% del tiempo. Esto quiere decir que las operaciones que se realizan en ese caso requiere mucho tiempo de cálculo. Concretamente, esas operaciones consisten en recorrer la lista de nodos para eliminar la descendencia de un nodo específico. Cuando la lista de nodos alcance grandes valores, tardará mucho en recorrerla.

5. Notas del alumno

5.1. Partes extras de la práctica

La práctica se compone de una parte obligatoria y una parte optativa. La parte obligatoria consiste en implementar el algoritmo Alfa-beta y redactar una documentación. En la parte optativa se han añadido determinadas funcionalidades que se describe en los puntos siguientes.

5.1.1. Implementación del algoritmo SSS*

Los detalles de la implementación de SSS* ya se han visto en apartados anteriores.

5.1.2. Límite de tiempo máximo para calcular el movimiento

El movimiento que la IA decida realizar debe obtenerse en un tiempo determinado para que la partida sea fluida y el jugador humano no sufra las pausas.

Según la cantidad de niveles que deben explorar los algoritmos, más tiempo tarda en completarse la ejecución. Además, dependiendo de la situación de la partida, habrá más movimientos o menos movimientos, lo que influye en el número de nodos del árbol que se generarán. Ambos factores hacen que el tiempo de ejecución sea indeterminado o de cierto grado de azar. Si el límite de tiempo es de 1 segundo y el algoritmo no ha encontrado un movimiento en ese tiempo, la IA no será capaz de mover ninguna ficha.

Este problema puede solucionarse explorando sólo hasta un nivel del árbol relativamente pequeño, de manera que el tiempo de ejecución del algoritmo sea mínimo. Sin embargo, al explorar menos nodos del árbol, es menos probable que el movimiento escogido sea de los mejores, lo que convierte a la IA en un rival débil.

La mejor solución al problema es calcular el movimiento a realizar modificando la cantidad de niveles que explora el algoritmo, comenzando desde el nivel 1 e incrementando de nivel sucesivamente. De esta manera, se calculará el movimiento varias veces en distintas dificultades. Cuando se rebase el límite de tiempo máximo, se detendrá la ejecución en curso y el último movimiento calculado por la última ejecución del algoritmo será el movimiento que realizará la IA. Siguiendo este proceso, nos aseguramos que la IA encuentre un movimiento siempre y, si le sobra tiempo, intentará encontrar un movimiento mejor.

La implementación en el código de la práctica consiste en un hilo de ejecución que recorre un bucle y calcula el movimiento repetidamente, partiendo desde el nivel 1 e incrementando un nivel en cada iteración[4]. Cuando se alcanza el límite de tiempo, se detiene la ejecución del hilo y el último movimiento calculado es el movimiento final. Este límite de tiempo puede modificarse desde la interfaz de usuario. Si este tiempo indicado no es suficiente para calcular al menos un nivel, se elegirá el primero de los movimientos.

5.1.3. Elegir qué algoritmo utilizar para calcular el movimiento

Con el fin de probar el funcionamiento de los algoritmos, se ha introducido la posibilidad de elegir entre los tres algoritmos eligiendo alguna de las tres casillas.

5.1.4. Indicar qué cantidad de niveles explorará el algoritmo seleccionado

Al igual que la funcionalidad anterior, se añade la posibilidad de modificar el número mínimo

4 En realidad, comienza en el nivel NIVEL_MINIMO y termina, como mucho, en el nivel NIVEL_MAXIMO, que son variables estáticas de la clase *JugadorMaquina* y se pueden modificar desde la interfaz de usuario.

y máximo de niveles que explorará el algoritmo, permitiendo cambiar la dificultad de la IA.

5.1.5. Sugerencia de movimiento cuando juegan humanos

Al pulsar el botón «Sugerir movimiento» se marca en la pantalla qué movimiento realizaría la IA si fuera ella la que tiene que mover. Sólo calcula el movimiento tal cual se ha explicado en apartados anteriores, pero sin efectuar el movimiento.

5.2. Secciones de código importantes

La tabla siguiente indica qué métodos y partes del código podría desear ver el profesor:

movimientosPosibles	Este método de la clase *Tablero* genera una lista –*ArrayList*– de movimientos. Se encuentra en la línea 570 del fichero *Tablero.java*.
funcionEvaluacion	Hay 4 funciones de evaluación en el fichero *Tablero.java*. A partir de la línea 732 están las 4 funciones. Son métodos de la clase *Tablero* que evalúan el tablero en cuestión y devuelven dicho valor.
alfaBeta, minimax, SSS	Estos métodos son la implementación de los algoritmos. Son métodos de la clase *JugadorMaquina*. Se encuentran en la línea 174, 262 y 330 respectivamente.
run	Es el método que se procesa al ejecutar un hilo. Se encarga de llamar a alguno de los algoritmos para calcular el movimiento una cantidad de niveles determinada desde la interfaz de usuario. Línea 42 del fichero JugadorMaquina.java.
pruebasEvaluacion	Es el método de la clase *JugadorMaquina* con el que se han realizado las distintas pruebas de experimentación del apartado 4. Se encuentra en la línea 81.