Memoria de la práctica final

Johanna Capote Robayna Guillermo Galindo Ortuño
Cristina de La Torre Villaverde

enero 2017

1 Objetivo

Nos piden realizar una estructura para representar una partida del juego Conecta 4. En concreto, un árbol de tableros donde se ven reflejadas todas las posibles jugadas a partir del tablero principal. Y sobre esta estructura, crear un jugador automático, capaz de decidir la mejor jugada analizando los tableros posteriores.

2 Clase JugadorAuto

Para afrontar el problema decidimos crear una clase *JugadorAuto* que consta de dos datos miembro y de varios métodos, que podemos clasificar en constructores, métodos para rellenar el árbol, métodos de búsqueda, métodos de actualización y métricas.

2.1 Datos miembro

En una primera versión planteamos que esta clase tuviera solo como dato miembro un *ArbolGeneral* de *Tablero*, en esta versión la profundidad del árbol se decidia a la hora de rellenar el árbol. Sin embargo, al cambiar el planteamiento de las funciones *rellenarNodo* y *rellenarNodoProfundidad* y hacerlas recursivas, tuvimos que añadir un dato miembro llamado *profundidad_max* para saber cuando parar la recursividad y salirnos de la función. Asi que este nuevo dato *profundidad_max* señala la altura actual del árbol y es muy útil a la hora de rellenar o actualizar el árbol de forma recursiva.

Depués de hacer las métricas nos surje un problema, llega un momento en la partida en la que el árbol no tiene el mismo número de hijos por rama, es decir, hay algunas ramas que tienen menos hijos porque alguna de sus columnas está llena. Esto hace que nuestras métricas a la hora de calcular que posición devolver, cometan un fallo, ya que interpretan que el tablero más a la izquierda se corresponde con insertar la ficha en la posición 0, sin embargo, puede ser que nos encontremos en el caso de que la columna 0 esté llena, por lo tanto, este primer tablero se correspondería con "insertar ficha en la columna 1", provocando así

un fallo con el entero devuelto. Para solucionarlo, cambiamos los datos miembro e implementamos un *ArbolGeneral* de *pair* formado por un *int* que indica que posición es la que se ha insertado y un *Tablero*.

2.2 Constructores

Desde el principio nos planteamos hacer dos constructores, el constructor vacío y un constructor que se le pasará un *Tablero* por referencia.

Constructor vacío: Decidimos implementar este constructor para evitar posibles problemas a la hora de crear un objeto de esta clase sin parámetros. Este constructor crea un árbol vacío de tableros (un árbol formado solo por una raíz con un *Tablero* vacío) y al dato *profundidad_max* le asigna el valor 0.

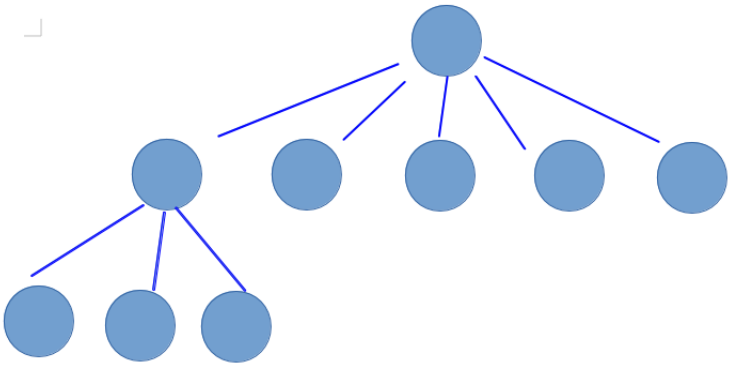
Constructor con un tablero por parámetro: Este constructor nos pareció muy útil a la hora de crear un objeto de la clase con un *Tablero* como raíz del árbol, así no tendríamos la necesidad de partir de un tablero vacío, sino que también podríamos empezar a calcular las futuras jugadas a partir de una partida ya empezada.

2.3 Rellenar

En una primera versión, decidimos solo implementar un único método llamado *rellenarArbol* en la cual se le pasaba un entero que indicaba la profundidad hasta la que se deseaba llegar. Esta versión, nos dió muchos problemas, al ser muy compleja y con muchas líneas de código. Por lo tanto decidimos dividirla en dos partes, *rellenarNodo* y *rellenarNodoprofundidad*.

rellenarNodo: a este método se le pasa un nodo, y a partir de éste se le añaden tantos hijos como jugadas posibles tenga el jugador al que le toca jugar.

rellenarNodoprofundidad: este método ha tenido dos versiones, una sin recursividad, y la definitiva, con recursividad. En la primera versión, intentamos prescindir de la recursividad mediante un *bucle for* en el que empezábamos recorriendo el árbol desde la raíz y le añadíamos el siguiente nivel, llamando a la función *rellenarNodo*, y posteriormente descendiendo un nivel y empezando desde el *hijoizquierda*, y continuando con el *hermanoderecha*. Más adelante nos dimos cuenta de que apartir del segundo nivel, este bucle solo iteraba entre los *hijosizquierda* y perdia la referencia de los *hermanosderecha*, es decir se nos quedaba un árbol con una forma parecida a la del dibujo.



Para arreglar el problema del método anterior, decidimos hacer la función recursiva, y para saber cuando parar la recursividad esta se va llamando a ella misma cada vez con una *profundidad* menor, de forma que cuando llegue a 0, ya se salga del método. En cada iteración, primero se comprueba que la *profundidad* es mayor que 0, en caso contrario sale del método,después se rellena el nodo actual, es decir, justo un nivel más bajo, llamando a *rellenarNodo* y por último se realiza un bucle en el que va recorriendo desde el hijo más a la izquierda hasta el hijo más a la derecha y llamando al metodo *rellenarNodofundidad* con *profundidad-1*.

2.4 Busqueda

Al principio no nos planteamos la necesidad de buscar en el arbol un tablero, pero a medida que se iban haciendo movimientos, nuestro árbol iba creciendo por lo que vimos útil implementar un método que actualizara el tablero y pusiera como raíz el tablero actual, es decir el instante donde se encuentra la partida. Para ello primero teniamos que buscar el tablero de la partida en nuestro árbol de tableros, así que siguiendo la filosofía de los métodos de rellenar, creamos dos métodos *buscarNodo* y *buscarNodoprofundidad*.

buscarNodo: a este método se le pasa como parámetro un *Tablero* que es el tablero que estamos buscando y un *Nodo*, que es el nodo padre del cual buscamos nuestro *Tablero* entre sus hijos.

buscarNodoprofundidad: este método sigue la misma filosofía recursiva del método *rellenarNodoprofundidad*. Primero comprueba que la profundidad es mayor que cero, después busca el *Tablero* en los hijos del nodo que se le pasa como parámetro y por último realiza un bucle empezando por el hijo más a la izquierda del Nodo y terminando por el hijo más a la derecha llamando en cada nodo a la función *buscarNodoprofundidad* con *profundidad-1*. En caso de que se encuentre el tablero se sale del método devolviendo el Nodo en el que se encuentra nuestro Tablero, sino lo encuentra devuelve cero.

2.5 Actualizar

A la hora de pensar en como actualizar nuestro árbol de tableros, se nos ocurrio ir actualizando jugada a jugada para hacerlo más sencillo. Sin embargo nos parecia que esto limitaba mucho este método, asi que preferimos implementarlo buscando el tablero en todo el árbol colocando este como raíz.

actualizar: a este método se le pasa un *Tablero*, y lo primero que hace es buscar ese tablero en nuestro árbol, si lo encuentra, lo coloca en la raíz y llama a *rellenarNodoProfundidad* con *profundidad_max* para que se quede un aábol de la misma altura del que teniamos antes. Antes de rellenarlo, comprueba que la altura a la que queremos llegar es más grande que la altura en la que se encuentra el árbol, porque sino lo estariamos rellenando "dos veces". Si no encuentra el *Tablero* en el árbol, coloca el Tablero pasado como parámetro en la raíz y lo rellena como si fuera un árbol nuevo, llamando a *rellenarNodoprofundidad* con *profundidad = profundidad_max*.

2.6 Metricas

A la hora de decidir la estrategia que iba a seguir nuestro jugador automático, se nos ocurrieron varias ideas, que podemos distinguir en defensivas y ofensivas.

2.6.1 Defensivas

Pensamos que una de las mejores estrategias para hacer jugadas buenas es evitar que te ganen, de esta idea surgieron dos métricas: *metrica_defensiva_simple* y *metrica_defensiva*.

métrica defensiva simple: esta métrica es muy sencilla, en primer lugar busca en el siguiente nivel si hay algún movimiento en el que gana y si es así devuelve ese movimiento; sino es así busca en el siguiente nivel si el rival tiene alguna oportunidad de ganar, en este caso inserta la ficha en ese lugar impidiendo perder en el siguiente movimiento.

métrica defensiva: la métrica defensiva sigue la misma filosofía que la métrica defensiva simple. Primero busca si gana en el siguiente movimiento,en cuyo caso devuelve ese movimiento; en caso contrario cuenta el número de derrotas en las posibles jugadas siguientes y elige la que tenga menor número de derrotas.

2.6.2 Ofensiva

Al principio pensamos en implementar una "métrica ofensiva simple" y otra "métrica ofensiva" más compleja pero decidimos que la "métrica ofensiva simple" no tenía mucho sentido puesto que ya todas las métricas lo primero que

hacían era comprobar si podian ganar en la siguiente jugada, así que solo implementamos una *metrica_ofensiva*

métrica ofensiva: esta métrica lo primero que hace es, como todas las anteriores, buscar si entre las siguientes jugadas hay alguna en la que gane, si es así, devuelve esa jugada; sino cuenta entre los niveles siguientes el número de posibles victorias de cada jugada que puede realizar y devuelve la jugada que mayor número de victorias tenga en los tableros posteriores.

3 Opinion personal

Nos ha parecido una práctica muy interesante, aunque un poco compleja, por el tema de la recursividad en los árboles. Hemos tenido que reflexionar sobre las métricas y la estrategia que seguiría nuestro jugador automático, y tras varias pruebas, hemos comprobado que la métrica más fuerte es la defensiva. Nos hubiera gustado añadir más métricas que fueran más fuertes y complejas, pero la falta de tiempo debido a los exámenes nos han hecho abandonar la idea.