Práctica 3

Daniel Monjas Miguélez

Análisis del problema

La dificultad de este problema se trata de generar un algoritmo que juegue al 4 en raya con algunas consideraciones extra. Estas son que cada jugador realiza dos movimientos en lugar de 1 y que existe un movimiento extra, la bomba que no está en el juego clásico y que le permite al jugador opuesto eliminar fichas nuestras.

El algoritmo que hemos utilizado para que juegue es un poda alfa-beta con profundida máxima 8, lo que nos da lugar a un nuevo problema, generar una heurística que haga que nuestro algoritmo sea un rival difícil.

La dificultad para considerar dicha heurística nuevamente es que los jugadores hacen dos movimientos en lugar de uno y que existen movimientos que alteran el estado de nuestras fichas y que se deben tener en cuenta. La solución propuesta se explica a continuación.

Descripción de la solución

Poda alfa-beta

En primer lugar explicaré como he programado la poda alfa-beta.

Mi función poda alfa-beta devuelve un double y tiene como parámetros el un objeto de la clase `Environment` (es el tablero), el jugador cuyo movimiento vamos a evaluar, la profundidad dentro del árbol, la profundidad máxima (establecida en el enunciado a 8), un objeto `Environment::ActionType` y las cotas alfa y beta que se van a utilizar, inicializadas en la función think.

En primer lugar si nuestro tablero no permite realizar movimientos, es decir, `JuegoTerminado()==true` o la profundidad actual es la máxima se devuelve el valor calculado por mi heurística.

En caso contrario se define un vector de bool que se usará para obtener las acciones aplicadas desde el estado actual del tablero. Se guarda en la variable `acciones_posible` el número de acciones que se pueden realizar desde el estado actual. Se define una variable `ultima_accion` que se inicializa a -1 y que se utilizará para generar sucesivamente los correspondientes movimientos. También se define un variable de tipo double `comparacion` que se usará para almacenar de forma auxiliar el valor devuelto por la poda alfa-beta para un hijo. Por último se definen un objeto `Environment::ActionType` que se utiliza para pasarlo como parámetro en la llamada a poda alfa-beta de los hijo y un objeto `Environment` que almacenará al hijo.

Tras esto se comprueba si el jugador que mueve en el tablero que comprobamos coincide con el que pasamos como parámetro en cuyo se tratará de un nodo MAX, en caso contrario se tratará de un nodo MIN. A continuación, en un bucle `for` cuya condición es que el valor de `i` sea menor que el valor del número de acciones posible y que la cota inferior alpha sea menor que la cota inferior beta, se genera el tablero hijo consecuencia de la primera acción, se calcula su valor de poda alfa-beta y si este valor supera a la cota

inferior se almacena la acción que nos lleva al nodo hijo como acción a realizar. Una vez terminado el bucle, si este a finalizado por criterio de poda se devuelve beta, en caso contrario se devuelve alpha.

De forma análoga se actua para un nodo MIN. La única diferencia es que la acción a realizar se actualiza si el valor de poda alfa-beta del hijo es menor que la cota superior beta. Por otro lado si el bucle for sale por criterio de poda se devuelve alpha en lugar de beta y si no sale por criterio de poda se devuelve beta.

De esta manera la poda alfa-beta que he programado gana al ninja 1 como jugador 1 y como jugador 2 con la heurística `ValoracionTest`.

Heurística

En mi heurística al igual que en `ValoracionTest` si el ganador es el jugador que evaluamos devuelve 99999999.0, si es el jugador opuesto, devuelve -99999999.0, si es un empate devuelve 0 y en caso contrario nos devuelve el valor calculado por la heurística.

Mi heurística tiene como parámetros el jugador y el tablero (objeto de clase `Environment`) que evaluamos. Lo primero que se hace es definir una variable jugador opuesto, que contendrá el número del jugador opuesto. Lo segundo que hacemos es definir un objeto double `suma` inicializado a 0. A dicho objeto se le suma la puntuación obtenido por la función `ProbVictoriaHorizontal`, la función `ProbVictoriaVertical` y la función `ProbVictoriaDiagonal`, por el jugador que estamos evaluando, y restándole lo obtenido en dichas funciones para el jugador contrario.

ProbVictoriaVertical

La función `ProbVictoriaVertical` tiene como parámetros al jugador que se evalua, al jugador opuesto y el estado actual. Se crea una variable double puntuación inicializada a 0, que será la que devolvamos. También se crean dos variables int `contador_jugador` y `contador_opuesto` ambas inicializadas a 0.

Tras esto se recorren todas las casillas entre las filas 1 y 4, ambas inclusive. Para cada una de estas casillas se comprueban quien ocupa dicha casilla y las 3 superiores, si la ocupa el `jugador` que evaluamos se aumenta `contador_jugador` y si la ocupa `jugador_opuesto` aumentamos `contador_opuesto`. El objetivo de este comprobación es ver cuantas posibilidades de hacer un 4 en raya vertical hay (solo se considera una posibilidad válida, si en las cuatro casillas alineadas hay al menos una casilla ocupada por jugador o por jugador opuesto, en caso contrario se omite dicha posibilidad), luego un posible cuatro en raya se descarta si se ha encontrado en el mismo alguna casilla ocupada por el jugador opuesto.

En caso de que no se haya encontrado se le suma a puntuación 2 puntos si `contador_jugador` es 1, 5 si `contador_jugador` es 2 y 10 si es 3.

Análogamente, para tener en cuenta al `jugador_opuesto`, si en dicha posibilidad solo hay casillas libres o casilla ocupadas por el `jugador_opuesto` se le resta a puntuación 2 si solo hay una casilla ocupada por `jugador_opuesto`, 5 si son 2 y 10 si son 3.

El proceso anterior se repite para todas y cada una de las casillas analizadas, y una vez terminada una casilla se reinician los contadores.

Finalmente, antes de devolver el valor puntuación se comprueba si hay bombas en el tablero, y dependiendo de quien sea la bomba se suma o se resta 10 puntos a la puntuación y se devuelve la puntuación.

ProbVictoriaHorizontal

El enfoque seguido para esta función es análogo al seguido para la función `ProbVictoriaVertical`, y los parámetros son lo mismo.

En este caso en lugar de considerar todas las casillas entre las fila 1 y la 4 se consideran todas las casillas entre las columnas 1 y 4. Y en lugar de comprobar la propia casilla y las 3 que están inmediatamente encima de la primera se comprueban las 3 que están inmediatamente a la derecha de la que se comprueba.

El criterio seguido para asignar la puntuación es exactamente el mismo que se ha seguido para el caso anterior. Si hay un posible 4 en raya sin que haya fichas del jugador opuesto de por medio se suma 1 punto si hay una casilla ocupada por el jugador, 5 si son 2 y 10 si son 3.

Análogamente, si hay un posible 4 en raya del jugador opuesto sin que haya casillas del jugador de por medio se resta 1 punto si hay una casilla ocupada por el jugador, 5 si son 2 y 10 si son 3.

Antes de devolver la puntuación se vuelve a hacer la comprobación de la bomba y por último se devuelve dicha puntuación.

ProbVictoriaDiagonal

Una vez más se repite el proceso seguido para las dos funciones anteriores. En este caso el proceso de ha divido en dos. Los primeros bucles comprueban las posibilidades de 4 en raya por medio de una diagonal creciente hacia la derecha, y los siguientes bucles comprueban las posibilidades de 4 en raya por medio de una diagonal creciente hacia la izquierda.

Para cada una de estas comprobaciones el recuento de puntos se hace de forma igual a las anteriores. Si hay una diagonal de fichas donde la primera ficha es de jugador, y en la diagonal no hay fichas de jugador opuesto, se suma 1 punto por una casilla de jugador, 5 si son 2 y 10 si son 3.

En caso de que la diagonal sea de jugador opuesto y no haya casillas de jugador en la misma, se resta 1 punto por una casilla de jugador, 5 si son 2 y 10 si son 3.

Por último se hace la comprobación de las casillas bomba y se devuelve el valor de la heurística.