UNIVERSIDAD DE GRANADA

INTELIGENCIA ARTIFICIAL

MEMORIA PRACTICA 3

MANCALA

ANTONIO GARCÍA CASTILLO

Diseño del estado

Para la búsqueda del mejor movimiento en cada ocasión a lo largo del juego uso un árbol de nodos implementado a través de un vector de la stl. Estos nodos son de tipo struct, declarados al principio del archivo BotKaLimon.h, y están compuesto por 6 componentes:

- ***GameState estado :*** Este campo es de tipo GameState y esta facilitado por la escuela. Nos permite en cada ocasión disponer de una serie de funciones que facilitan bastante el conocimiento del estado del juego. Puesto que no es de elaboración propia no nos detendremos mucho en este campo. He necesitado de este campo especialmente para el cálculo de la heurística, pues es necesario conocer quien es el jugador actual en ese nodo en particular, así como las fichas acumuladas por cada jugador en ese instante del juego y.

- ***Int movimiento :*** Este campo es necesario para saber cual es el movimiento o acción que hemos realizado desde el nodo padre para llegar a nuestro nodo actual. Obviamente esta acotado a valores entre 1 y 6. Recurro a este campo en dos ocasiones, cuando creamos los hijos en la función alfaBeta y obviamente cuando necesitamos devolver el movimiento que nos ha llevado a nuestro mejor hijo.

- ***Int padre :*** Nos indica el indice en el vector, nuestro árbol, de nuestro nodo padre desde el cual hemos sido creados. Sabiendo la posición de nuestro padre en el vector podemos saber si se ha repetido turno desde el ultimo movimiento, cosa que uso en el cálculo de la heuristica para premiar los nodos en los que hemos conseguido un turno extra.

- ***Int nivel :*** Con este campo controlamos la profundidad por la que vamos en nuestro árbol, es totalmente necesario controlarlo pues disponemos de un tiempo limitado para encontrar el mejor movimiento y nuestro factor de ramificación es de 6, lo que significa que nuestro árbol va a crecer bastante rápido y no vamos a poder explorarlo todo.

- ***Int mejorHijo :*** Guardaremos el indice en el árbol de nuestro hijo con mejor heurística en este campo, de esta manera podremos recorrerlo y obtener fácilmente en movimiento que nos ha llevado a él. Este campo se actualizará cuando encontremos un alfa o un beta mejor que el anterior, dentro de nuestra función alfabeta.

- ***Int valor :*** En este campo almacenaremos la heurística calculada. Se irá actualizando conforme el árbol vaya creciendo en función de si es nodo min y max, siempre y cuando cumpla las condiciones para ello, si es max que supere al alfa actual y si es min que sea menor que beta.

A continuación muestro exactamente la implementación tal y como esta en el archivo .h :

```
struct nodo
{
    GameState estado;
    int movimiento;
    int padre;
    int nivel;
    int mejorHijo;
    int valor;
};
```

Diseño del algoritmo

He basado mi práctica en el algoritmo minimax con poda alfaBeta, estableciendo una frontera a una profundidad de 12 niveles. Antes de iniciar la función recursiva hemos de declarar un nodo raíz inicializado debidamente con valores neutros y el estado de juego correspondiente. La cabecera de la función principal es la siguiente:

alfaBeta(vector<nodo> &arbol, int &indiceActual, int alfa, int beta);

Aunque puede resultar obvio, no esta de más comentar los parámetros de la función. El árbol de nodo pasado por referencia que se va expandiendo, el indice en el vector del nodo creado y nuestros valores alfa y beta con los que basamos nuestro algoritmo de búsqueda. El esquema seguido para la implementación del algoritmo es el siguiente:

- Si estoy en un nodo hoja o estoy en un nodo de profundidad 12 llamaremos a la función calcularHeuristica, el cual nos actualizará nuestro campo reservado para la heurística, valor mediante un return, lo que significa que no continuamos.

- Si no se ha dado el caso del anterior punto obviamente debemos seguir explorando el árbol, para ello crearemos todos los hijos del nodo actual mediante la función simulateMove(), facilitada por la escuela.

- En este punto hago una pequeña comprobación extra, pregunto si el estado creado es valido y si el movimiento que estamos simulando es sobre una casilla donde hay semillas, obviamente no podemos efectuar ninguna acción sobre una casilla vacía.

- Hemos avanzado, nuestro movimiento es totalmente válido, el siguiente paso es actualizar los parámetros de nuestro nuevo nodo, movimiento padre y profundidad.

- Aunque es conocido el funcionamiento del algoritmo alfaBeta, es necesario comentar que es en este momento cuando se produce la llamada recursiva, al estar todavía en el primer hijo creado se produce un avance en profundidad hasta que la primera condición de nuestro algoritmo se cumpla y devuelva un valor que actualice nuestros parámetros con los que trabajar a continuación.

- La parte final de mi algoritmo consiste en identificar si estoy en un nodo max o en un nodo min, ya que los turnos de los jugadores pueden repetirse no podemos obtener quien es el jugador actual por el simple hecho de estar en profundidad par o impar, para ello disponemos de una función de gameState que soluciona el problema. GetCurrentPlayer(). Si el jugador actual soy yo significa que necesitamos maximizar beneficio, por lo que actualizaremos mejor hijo, alfa y valor siempre que el valor devuelto por nuestra funcion recursiva sea menor que alfa y al contrario si nos encontramos en un nodo a minimizar. Basta decir que si alfa es mayor o igual que beta en cualquier nodo, esa rama se podará.

- El último paso es devolver el valor si no hemos podado, basta identificar quien es el jugador en ese nodo y devolver alfa si necesitamos maximizar, o beta si es nuestro rival el jugador en ese nodo.

Diseño de la heurística

Para el cálculo de la heurística me he basado en tres componentes básicas :

- Las semillas que llevo en mi granero en ese momento dado del juego. Entendemos que el objetivo del juego es conseguir más semillas que el contrario en tu propio granero, primaremos por encima de todo que en estados avanzados del juego nuestro granero tenga mas semillas que el del riva.

- Semillas del rival. Obviamente debemos minimizar el número de semillas que nuestro rival esta consiguiendo, dado que el juego ofrece 48 semillas exactamente, cuantas mas semillas consiga nuestro rival menos podremos conseguir nosotros. Debemos penalizar los estados del juego donde nuestro rival haya conseguido reunir más semillas que nosotros.

- Turno extra. Podemos conseguir un turno extra en varias situaciones del juego. En vez de buscar las casillas donde pudiera conseguir turno extra y aumentar la heurística en esos casos, me he decantado por premiar las situaciones donde ya se ha conseguido duplicar el turno extra y tendremos mas posibilidades de reunir mas semillas en nuestro granero. Cuando hemos conseguido doblar turno otorgo un valor numérico extra de 10 que sera añadido a la heurística final

El cálculo de la heurística final viene dado por la ecuación:

heurística = MisSemillas + turnoExtra – SemillasRival.

Sabemos que el objetivo del juego es lograr más semillas que el rival, que las semillas son exactamente 48 y que una vez nuestras semillas están en nuestro granero el rival no nos las puede quitar. Por tanto si se da la situación a lo largo del juego que nosotros o nuestro rival tiene más de 24 semillas en el granero sabremos con total seguridad que será el ganador final. Sabiendo esto, no tiene sentido calcular valores heurísticos de ese nodo si ya sabemos quien será el ganador, esta comprobación la hago al principio de la función heuristica devolviendo un valor muy alto si hemos conseguido superar la cantidad, forzando la actualización de alfa y que sea nuestro camino el elegido. Si es nuestro rival el que ha superado dicha cantidad, devolveremos un valor muy bajo, haciendo que beta se actualice y evitemos tomar ese camino en la medida de lo posible.