PRACTICA 3

IA

Jose Antonio Padial Molina
josepadial@correo.ugr.es

Contenido

Descripción del juego

Los componentes del juego son:

- El tablero está compuesto por una serie de receptáculos (que pueden ser agujeros hechos en el suelo o en un bloque de madera) organizados en filas.
- Estos receptáculos contienen una serie de piezas -48- (semillas, guijarros, canicas) indiferenciadas que son los elementos móviles del juego.
- El movimiento básico es el que habitualmente se llama "de siembra", que consiste en tomar todas las piezas contenidas en uno de los receptáculos e irlas depositando de una en una en receptáculos consecutivos a partir del siguiente al que las contenía.

Explicación de la composición del código

Nuestro programa está compuesto por una función donde se calculan todos los posibles futuros movimientos de forma recursiva y otra donde se les da puntuación a esos posibles movimientos para luego hacer una comparación.

Memoria de cómo se ha ido desarrollando la heurística

```
for(int i=1; i<7; i++){
    cantidad = state.getSeedsAt(oponente, (Position)(7-i));
    cantidad_mia = state.getSeedsAt(miTurno, (Position)(i+cantidad));
    if(cantidad < (7-i)){
        if(state.getSeedsAt(oponente, (Position)(7-i-cantidad)) == 0 && cantidad_mia != 0){
            if(cantidad > cantidad_ant_rob){
                hay_vacio = true;
                numero = (i+cantidad);
                cantidad_ant_rob = cantidad;
            }
        }
    }
}
```

Con este fragmento de código estoy buscando y dando más prioridad a los movimientos que más cantidad de piezas puedo obtener haciendo un robo al adversario.

```
for(int i=1; i<7; i++){
    cantidad_mia = state.getSeedsAt(miTurno, (Position)i);
    cantidad = state.getSeedsAt(oponente, (Position)(7-(i+cantidad)));
    if(cantidad_mia < i && cantidad_mia > 0){
        if(state.getSeedsAt(miTurno,(Position)(i-cantidad_mia)) == 0 && cantidad_ant < cantidad){
            if(state.getSeedsAt(oponente, (Position)(7-(i+cantidad))) > 0){
                tengo_que_robar = true;
                numero_mio = i;
                cantidad_ant = cantidad;
            }
        }
    }
}
```

Con este bucle hago el caso opuesto doy prioridad a los movimientos en los cuales evito que el adversario me quite una gran cantidad de piezas.

Con estos bucles así puedo valorar si es mejor robar o evitar que me robe según la situación del tablero.

```
for (int i = 1; i < 7; i++) {
    if (state.getSeedsAt(miTurno, (Position)i) == i) {
        movimiento = (Move)i;
        break;
    }
}
```

Y por último antes de entrar en casos más generales este bucle nos da mucha prioridad a los movimientos en los cuales dejamos la última pieza en granero.

```
for(int i = 1; i<7; i++){
    mis_puntos += st.getSeedsAt(player, (Position)i);
    sus_puntos += st.getSeedsAt(oponente, (Position)i);
}

valor = (st.getScore(player) - st.getScore(oponente)) + (mis_puntos - sus_puntos);
```

Para finalizar la heurística de mi bot tenemos el cálculo de las piezas que tengo en mi granero menos las del rival y a ese cálculo le sumo las piezas que tengo en mis casillas menos las del rival.

Función MiniMax

```
int MiniMax(Player player,GameState st, Move &mv, int profundidad){
    Move mierda;
    int mejor=INT32_MIN;
    int max = INT32_MIN;
    int min = INT32_MAX;
    if(st.isFinalState() || profundidad == 0)
        //return Heuristica(player, st);
        return st.getScore(player);
    else{
        for(int i=1; i<7; i++){
            if(st.getSeedsAt(player, (Position)i)){
                GameState gs = st.simulateMove((Move)i);
                int valor = MiniMax(player, gs, mierda, profundidad-1);

                if(st.getCurrentPlayer() == player){
                    if(valor > max){
                        max = valor;
                        mejor = valor;
                        mv = (Move)i;
                    }
                }
                else{
                    if(valor < min){
                        min = valor;
                        mejor = valor;
                        mv = (Move)i;
                    }
                }
            }
        }
        return mejor;
    }
}
```

Un árbol de juego es un árbol de posibles movimientos futuros. Cada nodo en el árbol son movimientos posibles, y los hijos de un nodo son movimientos posibles del oponente. Cada hoja del árbol recibe un puntaje que evalúa cuánto valdrá la posición para usted. Esto puede ser una evaluación absoluta, como la victoria de un juego, o confiar en alguna heurística, como las piezas actuales por cada lado. El puntaje luego pasa por el árbol a través de la búsqueda minimax. En la búsqueda minimax, eliges el puntaje máximo para cada una de las ramas de tu giro y asumes que tu oponente elegirá el movimiento que minimiza tu puntaje para cada uno de sus turnos. El movimiento real en cada turno será el movimiento en la parte superior del árbol con la puntuación minimax más alta.

Función Poda Alfa-Beta

Como último cambio realizado al código se implementó esta función y se anuló la función MiniMax.

```
int Poda(Player player, GameState st, Move &mv, int profundidad, int alfa, int beta){
    int mejor;

    if(st.isFinalState() || profundidad == 0){
        return Heuristica(player, st);
        //return st.getScore(player);
    }
    else{
        if(player == st.getCurrentPlayer()){
            for(int i=1; i<7; i++){
                if(st.getSeedsAt(player, (Position)i)){
                    GameState gs = st.simulateMove((Move)i);
                    Move  accionAnterior;
                    mejor = Poda(player, gs, accionAnterior, profundidad-1, alfa, beta);
                    if(gs.getCurrentPlayer() == player) mejor = 500;

                    if(alfa < mejor){
                        alfa = mejor;
                        mv = (Move)i;

                    }
                    if(beta <= alfa){
                        return alfa;

                    }
                }
            }
            return alfa;
        }
        else{
            for(int i=1; i<7; i++){
                if(st.getSeedsAt(player, (Position)i)){
                    GameState gs = st.simulateMove((Move)i);
                    mejor = Poda(player, gs, mv, profundidad-1, alfa, beta);
                    if(gs.getCurrentPlayer() == player) mejor = 500;

                    if(beta > mejor){
                        beta = mejor;
                        //mv = (Move) i;

                      }
                      if(beta<=alfa){
                          return beta;
                      }
                  }
              }
             return beta;
         }
     }
}
```

Existe un problema en el Algoritmo Minimax, el cual tiene que ver con la complejidad del mismo, ya que el número de estados a examinar o evaluar es exponencial en el número de movimientos. Sin embargo, aunque este exponente no se pueda eliminar, existe una posibilidad de reducirlo a la mitad, y esto mediante la realización de una Poda al árbol Minimax, en nuestro caso: Poda Alfa-Beta, donde Alfa y Beta representan, respectivamente, las cotas inferior y superior de los valores que se van a ir buscando en la parte del subárbol que queda por explorar. Si Alfa llega a superar o igualar a Beta, entonces no será necesario seguir analizando las ramas del subárbol.