Rocío Barragán Moreno 3º Ingeniería informática y ADE.

MEMORIA PRÁCTICA 3 INTELIGENCIA ARTIFICIAL

El objetivo principal de esta práctica ha consistido en derrotar a 3 Ninjas (cada uno a un nivel más avanzado) en el famoso juego del Parchís. Sin embargo, este juego no cumplía con las mismas reglas que el parchís original, ya que sino no podríamos trabajar con el agente activo deliberativo aprendido en esta asignatura.

Para abordar esta práctica, era necesario implementar el algoritmo Minimax usando la poda Alfa Beta y también una heurística que determine el peso de los nodos del árbol del juego, dependiendo de lo que consideramos beneficioso para nosotros (Jugador principal) y contra cada uno de nuestros oponentes (Los Ninjas).

1. Algoritmo de Poda Alfa-Beta

Para implementar este algoritmo, me he apoyado en pseudocódigos que he ido encontrando, ya que es bastante parecido generalmente.
La función principal de este algoritmo es devolver el próximo movimiento que se va a realizar teniendo en cuenta la heurística implementada, la cuál se le pasa por parámetro.
Un detalle que quería destacar acerca de la implementación que he hecho, es que para generar el siguiente movimiento he usado el método ***`generateNextMoveDescending()`*** en concreto porque se irán generando los hijos empezando por el mayor, que es el que nos interesa realmente, ya que normalmente preferimos movernos con los valores más altos del dado para ir avanzando más.
En general no me ha supuesto mucho esta implementación, ya que es un método bastante común y similar en la mayoría de casos.

2. Algoritmo de la Heurística

Al principio, comencé implementando una heurística que tuviese en cuenta muchísimos casos a la hora de mover las fichas, pero por los resultados que obtuve decidí hacer otra mucho más simple, y al parecer es mucho más eficiente.

Mi heurística está dividida en 2 partes. Una calcula una especie de heurística para el jugador principal, y otra para el oponente. Al final cada una se irá sumando a la heurística principal que es la que nos interesa (variable ***`resultadoHeuristica`***).

→Heurística para el jugador principal.
La idea de esta heurística es básicamente beneficiar en 2 casos:

- Cuando el jugador se encuentre en una casilla de posición segura, ***`isSafeBox()`***.
- Cuando el jugador se encuentre en el carril de la meta final, en su carril de su mismo color y que por tanto está a salvo y ya no puede salir de ahí, va rebotando si saca números mayores a los que necesita para entrar en casa, ***`distanceToGoal()`***.

Y perjudicar en caso de que alguna ficha del jugador esté en casa, puesto que está lejos de la meta.

→ Heurística para el oponente

Es bastante parecida a la anterior, y representa cómo beneficia al ninja ese tablero que hemos representado antes. Por tanto, si la ficha del jugador principal está en casa, beneficia a la heurística lógicamente, así como que si la distancia del jugador a la meta es lejana, también le beneficiará pues tiene más posibilidades de ganar.
He añadido también otro método que perjudicaría a la heurística: ***`isEatingMove()`***, ya que si el jugador principal se come una ficha del oponente, le perjudicaría.

Y ya, como he mencionado anteriormente, estas heurísticas se le irían sumando a la heurística principal, que es la que verdaderamente nos interesa.
También cabe destacar que los números que he usado son un poco de manera aleatoria, fui probando números más grandes, más pequeños, y al final me quedé con estos porque eran los que me daban mejor resultado básicamente.

Heurística Jugador principal

```
double resultadoHeuristica = 0.0; //aquí se irá guardando el resultado final de la heurística.

vector<color> misColores = estado.getPlayerColors(jugador);

//Recorro los colores de mi jugador
 for(int i=0;i<misColores.size();i++){
    double heuristicaJugador = 0.0;
    color colorJugador = misColores[i];

    heuristicaJugador -= 25000*estado.piecesAtHome(colorJugador);

    //Recorro las fichas de ese color
    for(int j=0; j<num_pieces; j++){
        int distancia = estado.distanceToGoal(colorJugador,j);

        if(estado.isSafeBox(estado.getBoard().getPiece(colorJugador,j))){

            heuristicaJugador += 8000;
        }
        else if(estado.distanceToGoal(colorJugador,j)<=7){

            heuristicaJugador+=7000;
        }
        else{
            heuristicaJugador-=500*distancia;
        }
    }
    resultadoHeuristica += heuristicaJugador;
}
```

Heurística oponente.

```
int oponente = (jugador + 1) % 2;
vector<color> coloresOponente = estado.getPlayerColors(oponente);

//Recorro los colores del oponente
for(int i=0;i<coloresOponente.size();i++){
    double heuristicaOponente = 0.0;
    color colorOponente = coloresOponente[i];

    heuristicaOponente += 20000*estado.piecesAtHome(colorOponente);

    //Recorro las fichas de ese color también
    for(int j=0;j<num_pieces;j++){
        int distancia = estado.distanceToGoal(colorOponente,j);

        if(distancia>7){
                heuristicaOponente+=500*distancia;
        }
        else if(estado.isEatingMove()){
                heuristicaOponente-=2000;
        }
        else{
                heuristicaOponente-=1000;
        }

    }
    resultadoHeuristica += heuristicaOponente;
}

return resultadoHeuristica;
```

Para terminar la memoria, me gustaría comentar unas cosas acerca de los resultados del juego.

Cuando ejecuté el juego, me salió en principio que ganaba a todos, las 6 veces posibles. Pero lo fui probando más veces para asegurarme y a veces perdía en Ninja3 vs Heurística, o sea que ganaba el Ninja3. Como me daban estos resultados tan extraños decidí probar mi código en el ordenador de algunos compañeros, y a ellos siempre les salía que ganaba todas. Por tanto sigo teniendo cierta incertidumbre sobre si gana a todos o sólo a todos menos uno, pero por lo pronto ganaría a todos. No sé si ha sido problema de algo de mi ordenador o algo por el estilo.

En cualquier caso, en esta memoria se exponen todos los algoritmos y estrategias que he utilizado, que a mi parecer son bastante coherentes.