INTELIGENCIA ARTIFICIAL

PRÁCTICA 3:

Métodos de búsqueda con adversario

Desconecta-4 Boom

José Manuel Navarro Cuartero

Análisis del problema

En esta práctica se nos plantea el problema de diseñar un agente deliberativo para el juego “Desconecta-4 Boom” que utilice una poda alfa-beta. Del juego conocemos todas sus variables y escenarios posibles, luego podemos implementar un comportamiento completamente deliberativo. Además, como no tenemos memoria infinita, tendremos que implementar una heurística que será usada por la poda, para decidir qué movimientos son mejores.

El juego en sí, es una adaptación del conocido “Conecta-4”, pero con dos modificaciones que cambian drásticamente la forma de entenderlo. Por un lado, además de las fichas normales, cada cinco turnos se dispone de una ficha “boom”. Esta ficha elimina todas las fichas de su color en su fila. Por otro lado, cambia la finalidad del juego. En lugar de intentar enlazar 4 fichas consecutivas, en esta versión se trata de que sea el rival el que las enlace.

Estos dos cambios cambian el planteamiento que se hace a priori del juego, puesto que por un lado cobra más importancia el “no perder” que el ganar, a la vez que se intenta forzar al rival a que haga una mala jugada. Por otro lado, la variable de poder eliminar las fichas propias, moviendo las del rival que haya por encima, abre un abismo de posibilidades.

Descripción de la solución planteada

Se ha implementado una solución con una poda alfa-beta, y una heurística apoyada de 3 funciones auxiliares, que procederemos a describir.

Por un lado, en la función think(), al final de la misma se hace la llamada a la poda alfa-beta:

```
// Opcion: Poda AlfaBeta
// NOTA: La parametrizacion es solo orientativa
alpha=menosinf; beta=masinf;
valor = Poda_AlfaBeta(actual_, jugador_, true, PROFUNDIDAD_ALFABETA, accion, alpha, beta);
cout << "Valor AlfaBeta: " << valor << "  Accion: " << actual_.ActionStr(accion) << endl;
return accion;
```

Es esta función, Poda_AlfaBeta, decidimos usar la parametrización que se nos sugiere, a saber, el estado actual en el que nos encontramos, el jugador, un booleano (en nuestro caso, empezamos con true), la profundidad (en la primera llamada es la máxima, y va bajando hasta 0), la acción que se devolverá (pasa por referencia), y los valores alfa y beta (masinf y menosinf en la primera llamada).

La función de la poda es relativamente simple. A saber, primero comprueba que no se haya alcanzado la profundidad máxima, es decir, que el parámetro “prof” valga cero, o que el juego haya terminado. Si se dan cualquiera de ambas, acaba la función recurrente, y devuelve el valor de la heurística en el estado que lleve como parámetro.

Si el juego no ha acabado en el estado que consideramos, se generan todos los hijos posibles del estado, es decir, todos los movimientos que puede hacer el jugador que estén permitidos, y se guardan en variables auxiliares.

Se vuelve a llamar a la función entonces, pero esta vez, con el contrario del booleano en los parámetros, ya que el siguiente movimiento será del rival, y con la profundidad valiendo 1 menos.

Pasamos ahora a la heurística, la cual se podría dividir en dos partes. La primera parte es similar a la "ValoracionTest" que se nos daba. A saber, se comprueba si alguien ha ganado (para devolver un valor infinitamente positivo o negativo), o si el juego ha acabado en empate. Si no han ocurrido ninguno de ambos, se procede a aplicar la heurística propiamente dicha.

En este caso, y dada la similitud del juego con el "Conecta-4", se decidió hacer una heurística como si fuera para ese juego directamente. Es decir, se recorre el tablero en todas las direcciones posibles (a través de 3 funciones auxiliares, "Hor", "Ver" y "Dia"), y se valora positivamente que haya varias fichas del jugador en cuestión seguidas, con un hueco después en el que poder poner otra más.

```
double Valoracion(const Environment &estado, int jugador){
  int ganador = estado.RevisarTablero();

  if (ganador==jugador)
    return 99999999.0; // Gana el jugador que pide la valoracion
  else if (ganador!=0)
    return -99999999.0; // Pierde el jugador que pide la valoracion
  else if (estado.Get_Casillas_Libres()==0)
    return 0;  // Hay un empate global y se ha rellenado completamente el tablero
  else { // Aplicamos la heuristica
    int rival = (jugador==1)?2:1;
    int yo_hor = Hor(estado, jugador),  yo_ver = Ver(estado, jugador),  yo_dia = Dia(estado, jugador),
    rival_hor  = Hor(estado, rival), rival_ver = Ver(estado, rival), rival_dia = Dia(estado, rival);
    return ((rival_hor+rival_ver+rival_dia)-(yo_hor+yo_ver+yo_dia));
  } // Como las funciones suman en positivo las fichas seguidas, y nosotros buscamos que el rival las tenga,
}   // la heuristica hace rival-yo, para favorecer aquellos estados en los que el rival enlace fichas
```

Teniendo los 6 valores de éstas funciones tanto para el jugador como para el rival, simplemente se suman los del rival, y se les resta la suma de los del jugador. Es decir, se favorecen aquellos estados en los que el rival enlace varias fichas seguidas, y se desfavorecen aquellos para los que sea el jugador el que las enlaza.

Por último, en lo referente a las 3 funciones auxiliares, cada una de ellas recorre el tablero en una dirección (salvo "Dia", que hace las dos diagonales), y va comprobando si hay una ficha en cada uno de los huecos posibles. Si el jugador que se ha pasado en los parámetros tiene una ficha en ese hueco, aumenta una variable llamada "juntas", y se suman puntos al valor que se retornará, "puntos".

Si además, son varias las que se enlazan, el aumento en “puntos” es mayor, para favorecer las parejas o las ternas de fichas. Por otro lado, si inmediatamente después de una o más fichas del jugador se encuentra una del rival, se restan los puntos antes sumados, ya que de nada sirve que haya 3 seguidas si el hueco de la cuarta está ya relleno.

Aunque, si se revisan los números se verá que no se resta tanto como se suma. Esta decisión se tomó para tener en cuenta de alguna forma que gracias a los “boom” se pueden generar huecos en lugares donde antes había fichas, algo que no puede ocurrir en el “Conecta-4” original. La función para el recorrido horizontal sería:

```
for(int i=0;i<7;++i){ // Recorremos
  for(int j=0; j<7; ++j){
    if(estado.See_Casilla(i,j)==jug || estado.See_Casilla(i,j)==bomb){
      ++juntas;
      switch(juntas){
        case 2: puntos+=4; break;
        case 3: puntos+=12; break;
        default: puntos+=juntas; // 1 solitaria
      }
    } else if(estado.See_Casilla(i,j)==rival || estado.See_Casilla(i,j)==riv_bomb){
      switch(juntas){
        case 2: puntos-=4; break;
        case 3: puntos-=12; break;
        default: puntos-=juntas; // 1 o 0 (e.g. dos seguidas del rival)
      }
      juntas=0; // Se resetea juntas
    } else
      juntas=0; // Si esta vacio
  }
  juntas=0;
}
return puntos;
```

Las funciones auxiliares para el recorrido en vertical y en diagonal se harían de forma análoga a ésta.