UNIVERSIDAD
DE GRANADA

2ºC

Grado en Ingeniería Informática

Inteligencia Artificial:
Práctica 3 - Desconecta4BOOM .

Autor:
Antonio David Villegas Yeguas

Asignatura:
Inteligencia Artificial

4 de junio de 2019

1. Introducción.

Esta práctica consiste en diseñar e implementar una técnica de búsqueda con adversario en un entorno de juegos. En este caso se trata del Desconecta4BOOM, un juego con información completa en el que dos jugadores se enfrentan, tratando que el adversario conecte 4 fichas en vertical, horizontal o diagonal. Además, se añade el factor de las bombas, que cuando son explotadas todas las fichas del jugador que comparten fila con la bomba son retiradas del tablero.

Para la solución de esta práctica se nos daba a elegir entre la técnica MINIMAX o la poda Alfa-Beta, y además de implementar alguna de estas dos técnicas, implementar una función de evaluación heurística para evaluar como de beneficiosos son los distintos estados debido a que el árbol de movimientos posibles es demasiado grande como para explorarlo en la totalidad.

2. Decisiones sobre la técnica de búsqueda.

En un principio no tenía claro que método (MINIMAX o Poda Alfa-Beta) implementar, tenía claro que la Poda sería mucho mejor, ya que no exploraría la totalidad de los hijos, y me daría mejores resultados, ya que sería capaz de explorar el árbol con un mayor nivel de profundidad.

Tras consultar los materiales dados por los profesores de la asignatura, tanto las diapositivas de teoría, que contienen un pequeño pseudocódigo de como funciona la Poda, el libro Inteligencia Artificial: Un Enfoque Moderno de Stuart Russell y Peter Norvig, y la ayuda de los ejercicios resueltos en clases prácticas, me decidí a implementar la Poda Alfa-Beta, ya que fui capaz de entenderla y resolver los ejercicios propuestos, y en mi cabeza ya estaba la idea de como resolver la práctica.

3. Implementación de la Poda Alfa-Beta

Para la desarrollar la Poda, he seguido las explicaciones dadas por los profesores de la asignatura, y que hemos visto tanto en teoría como en prácticas. La forma de la Poda se resume en:

```
alfa = menosinf beta = masinf
Poda(estado, profundidad, alfa, beta):
Si nodo-terminal O profundidad = 0
    devolver evaluacion(nodo)
Generar hijos estado actual
Si le toca al jugador:
    Para todos los hijos:
        alfa = max(alfa, Poda(hijo, profundidad - 1, alfa, beta))
        si alfa >= beta devolver beta
    devolver alfa
Si le toca al oponente:
    Para todos los hijos:
        beta = min(beta, Poda(hijo, profundidad - 1, alfa, beta))
        si beta <= alfa devolver alfa
    devolver beta
```

Esto es solo una aproximación, ya que en nuestro código tenemos que tener en cuenta como actualizar la acción.

Cada vez que actualicemos alfa en un nodo MAX comprobaremos si estamos en el nodo raíz, y si esto se cumple, actualizamos la acción, ya que esto quiere decir que nos ha llegado un mejor valor para esta acción.

```
// si el nuevo valor es mayor que alfa, actualizamos alfa
if (v > alfa){
   alfa = v;

   // si estamos en el nodo raiz y hemos actualizado alfa quiere decir
   // que hemos encontrado un mejor camino a traves de la accion i
   // luego actualizamos la accion
   if (PROFUNDIDAD_ALFABETA == Player::PROFUNDIDAD_ALFABETA_){
     accion = static_cast< Environment::ActionType > (acciones[i].Last_Action(jugador_) );
  }
}
```

Además en nuestra función de valoración tenemos que tener en cuenta el estado que queremos valorar, el jugador con el que tenemos que encontrar el buen estado, y la profundidad desde el comienzo de la búsqueda.

```
// si llegamos a un nodo terminal, o al limite de profundidad
// devolvemos el valor de ese estado
if (PROFUNDIDAD_ALFABETA == 0 || actual_.JuegoTerminado()){
   return Valoracion(actual_, jugador_, PROFUNDIDAD_ALFABETA);
}
// si no, pasamos a generar sus hijos
```

En definitiva la poda nos permitirá evitar explorar grandes partes del árbol debido a que algunas ramas no importaran a la hora de tomar la decisión ya que existe un mejor valor y sabemos con certeza que esa decisión no será tomada. De esta forma, aunque el número de hijos siga siendo grande, mientras que usando MINIMAX teníamos el exponente del numero de hijos de un nivel elevado a la profundidad de nodos totales, usando la poda podríamos reducir ese número de nodos totales en la mitad.

En nuestro caso, el guión de prácticas nos ha impuesto un corte de la búsqueda de ocho niveles de profundidad, sin embargo, eso no me ha impedido que de forma didáctica compruebe la diferencia de usar distintos niveles de corte. Un claro ejemplo de esto es ver como usando una función de valoración bastante temprana, en la que apenas diferenciaba los estados y se basaba en contar fichas por filas, la diferencia de jugadas entre usar un corte de 4, 8 y 10, era bastante significativo, hasta tal punto de, una vez di por finalizada por completo mi función de evaluación, plantearme modificar la profundidad dada. Finalmente me decidí a no hacerlo, ya que una vez completa la función de valoración era capaz de ganar bastante rápido a los ninjas propuestos para la evaluación y sumando que una segunda lectura al guión de la práctica me confirmo que el máximo permitido para la Poda Alfa-Beta era 8.

Gracias a esta forma de búsqueda con adversario conseguimos evitar explorar todas las ramas, ya que en el el valor de alfa almacenaremos el valor de los nodos MAX y en beta el valor de los nodos MIN, de esta forma, cuando alfa $\geq$ beta quiere decir que la rama en la que se actualizo el valor de alfa será mejor, y nunca se llegara a escoger esta nueva rama.

4. Función de valoración

Mi función de valoración esta compuesta por dos partes.

4.1. Comprobación del ganador

Al comenzar esta función, comprobara si existe un ganador de la partida.

En un principio, si encontraba un ganador esta función devolvía $99999999{,}0$ si el ganador era el jugador o $-99999999{,}0$ si era el oponente. Conforme avanzaba con el desarrollo de la práctica observaba que en muchas situaciones, todas las jugadas posibles me llevaban a victoria, sin embargo, mientras que explotando la bomba ganaría con un turno, decidía realizar otras acciones, también validas porque llevaban a la victoria, pero que necesitaban más jugadas.

Esta situación la conseguí arreglar pasando el nivel de profundidad a la función de valoración, de esta forma, en lugar de devolver $99999999{,}0$ si ganaba, la condición de ganar esta llevada por el siguiente calculo $98999999{,}0 + (1000{,}0 * profundidad)$.

Vemos como he cambiado el valor total, para evitar que la valoración sea mayor que *masinf*, y dependiendo de la profundidad consideramos un estado mejor que otro, como la profundidad es decreciente desde 8 hasta 0, una profundidad más alta indica que hay que bajar menos niveles en el árbol, lo que indica una acción más próxima.

De la misma forma he modificado la parte en la que pierde el jugador, para que intente realizar el mayor número de jugadas posibles antes de perder.

```
// Funcion heuristica (ESTA ES LA QUE TENEIS QUE MODIFICAR)
double Valoracion(const Environment &estado, int jugador, int profundidad){

    // comprobamos si en el estado dado gana algun jugador
    int ganador = estado.RevisarTablero();

    if (ganador==jugador)
       // si la profundidad = 0 es que necesita realizar muchas acciones
      // para ganar, asi que preferimos otro estado a otro en el que gane
      // en menos jugadas
       return 98999999.0 + (1000.0 * profundidad);
    else if (ganador!=0)
            // si la profundidad = 0 es que necesita realizar muchas acciones
            // para perder, asi que preferimos ese estado a otro en el que pierda
            // en una jugada
            return -98999999.0 - (1000.0 * profundidad);
    else if (estado.Get_Casillas_Libres()==0)
            return 0;  // Hay un empate global y se ha rellenado completamente el tablero
    else
        return F_Valoracion(jugador,estado);// valoramos el estado para ver como de bueno es


}
```

Si no existe un ganador, pasamos a la segunda parte, hacer una estimación de como de bueno (o malo) es un estado.

4.2. Estimación del estado del tablero.

Para esta estimación lo que se realizará es recorrer la matriz del tablero, comprobando que casillas pertenecen a nuestro jugador, y que otras pertenecen al oponente.

```
double F_Valoracion(const int jugador, const Environment & estado){
    double h = 0;

    // valoramos el estado comprobando las piezas en vertical
    h += ValoracionVertical(jugador, estado);
    // valoramos el estado comprobando las piezas en horizontal
    h += ValoracionHorizontal(jugador, estado);
    // valoramos el estado comprobando las piezas en diagonal
    h += ValoracionDiagonal(jugador, estado);

    return h;

}
```

En cada una de estas llamadas, lo que cambiaremos es la forma de recorrer la matriz, pero la forma de asignar valores va a ser la misma.

Si una casilla es del jugador, le damos un valor (2 en mi caso) y aumentamos en 1 las casillas seguidas que tenia, así, en la siguiente iteración si la casilla anterior es del jugador, sumamos $4 * numero_casillas$, de esta forma, damos un mayor valor a las casillas contiguas que a las casillas que no tienen vecinos. Estos cálculos también se realizarán con las casillas del oponente.

Finalmente, en el valor total a devolver *h* sumaremos el valor de las casillas del oponente, ya que es bueno para el jugador que el oponente tenga casillas ocupadas, y restaremos las del jugador, ya que es malo que el jugador tenga casillas.

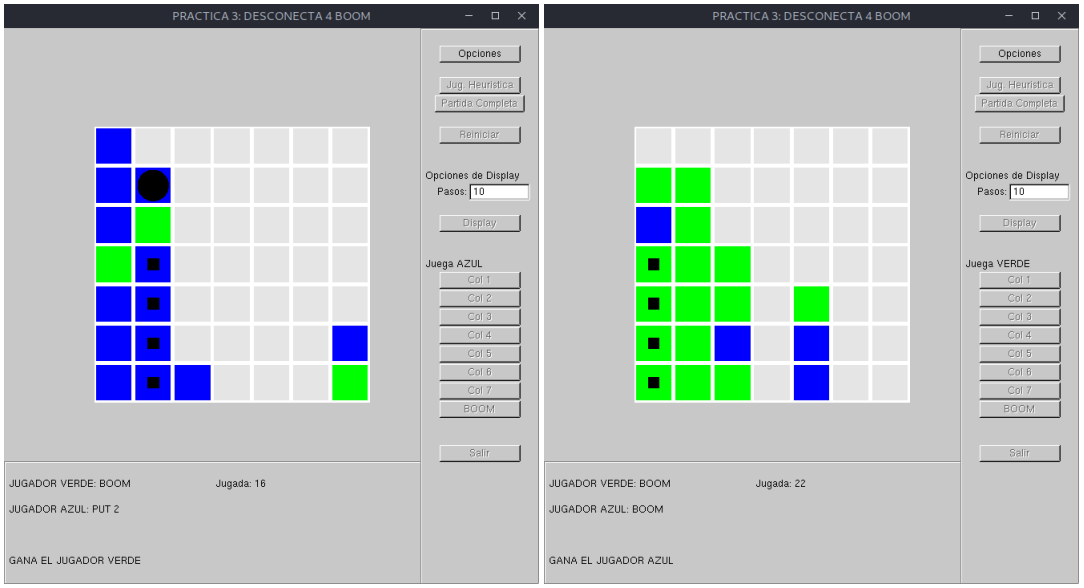
Estas operaciones las realizaremos recorriendo la matriz del tablero de forma vertical, horizontal, y diagonal.

De esta forma conseguimos los siguientes resultados contra los ninjas:

Ninja	Número de jugadas para ganar
Ninja 1: Jugador verde	21
Ninja 1: Jugador azul	22
Ninja 2: Jugador verde	16
Ninja 2: Jugador azul	22
Ninja 3: Jugador verde	21
Ninja 3: Jugador azul	17

Algunos ejemplo:

Contra el ninja de nivel 2, jugando mi heuristica como el jugador verde y como el jugador azul.



4.3. Conclusiones

Con esta práctica vemos el potencial de la búsqueda con adversarios aplicada a juegos, y como el tener una buena función de valoración es crucial ya que es imposible recorrer la totalidad del árbol de búsqueda y esta valoración nos ayudará a ser capaces de tomar un buen camino de decisiones.

Además me ha enseñado como el mismo algoritmo de búsqueda con adversarios puede aplicarse a todo tipo de juegos, ya que la distinción entre los juegos será la función de valoración