UNIVERSIDAD DE GRANADA

INTELIGENCIA ARTIFICIAL

E.T.S. de Ingenierías Informática y de Telecomunicación

Práctica 3: Métodos de Búsqueda con Adversario (Juegos)
DESCONECTA-4 BOOM

Autor: Francisco Javier Bolívar Expósito

Curso: 2018-2019

1. Análisis del Problema

Se pide diseñar e implementar un agente deliberativo capaz de jugar con un comportamiento inteligente al juego DESCONECTA-4 BOOM.

Para conseguir este comportamiento podemos implementar o bien el algoritmo de búsqueda en los estados que generemos del juego minimax o la versión con poda alfa-beta que permitirá alcanzar una profundidad mayor de búsqueda. También será necesario definir una heurística apropiada para este juego en concreto que al ser usada por el algoritmo proporcione un buen jugador artificial.

2. Descripción de la solución presentada

2.1. Implementación de la Poda ALFA-BETA

Se ha elegido entre las 2 opciones del algoritmo de búsqueda (minimax y poda alfa-beta) implementar el algoritmo con poda alfa-beta.

Código de la implementación:

```
double Poda_AlfaBeta(const Environment &t, int jug, int prof, int Limite,
Environment::ActionType &accion, double alpha, double beta)
{
	if (t.JuegoTerminado() || prof == Limite)
		return Valoracion(t, jug, prof);

	Environment v[8];
	int n = t.GenerateAllMoves(v);
	bool podar = false;

	for (int i = 0; i < n && !podar; ++i)
	{
		double aux = Poda_AlfaBeta(v[i], jug, prof+1, Limite, accion, alpha,
beta);

		if (t.JugadorActivo() == jug)
		{
			if (aux > alpha)
			{
				alpha = aux;
				if (prof == 0)
					accion =
static_cast<Environment::ActionType>(v[i].Last_Action(jug));
			}
		}
		else
		{
			if (aux < beta)
			{
```

```
                        beta = aux;
                  }
            }

            if (alpha >= beta)
                  podar = true;
      }

      if (t.JugadorActivo() == jug)
            return alpha;
      else
            return beta;
}
```

Esta función, usando la heurística que luego será definida, buscará recursivamente simulando las posibles acciones que puede realizar el jugador, pasando por nodos MAX (Jugador que usa la función) y MIN (Jugador contrario) hasta llegar a la profundidad máxima establecida (8 en nuestro caso) donde se evaluará la heurística. Los nodos MAX intentarán maximizar su valor y los nodos MIN minimizarlo y si no es necesario seguir buscando por una rama porque ningún valor va a cambiar el resultado escogido no se explora la rama. Para implementar esta búsqueda seguimos los siguientes pasos:

Al llamar por primera vez a la función le pasamos la cota alfa inicializada a menosinf (constante global del programa) y la cota beta inicializada a masinf(constante global del programa).

Si nos encontramos en un nodo terminal o en la profundidad límite evaluamos el nodo con la heurística y devolvemos este valor. En caso contrario, aplicamos cada movimiento posible sobre el estado actual y llamamos a la función recursivamente con el estado que genera, pasando las cotas alfa-beta del padre al hijo.

Si nos encontramos en un nodo MAX se modifica alfa cada vez que el valor que devuelva alguno de los movimientos analizados sea mayor que alfa actual y devolvemos alfa.

Si nos encontramos en un nodo MIN se modifica beta cada vez que el valor que devuelva alguno de los movimientos analizados sea menor que beta actual y devolvemos beta.

Si alfa es mayor o igual que beta se podan los hijos y no analizamos ninguno más.

Además añadimos que cuando nos encontremos en el nodo raíz y se encuentre una accion que genere un estado con una cota alfa mayor actualizamos también la acción.

De esta forma ya tenemos un algoritmo de búsqueda funcional que podemos probar con la función ValoracionTest proporcionada, pero será necesario definir una heurística mejor para obtener buenos resultados.

2. Definición de la heurística

Código de la heurística:

```
double Valoracion(const Environment &estado, int jugador, int prof){
      int ganador = estado.RevisarTablero();
    if (ganador==jugador)
       return (99999999.0 - prof); // Gana el jugador que pide la valoracion
    else if (ganador!=0)
       return (-99999999.0 + prof); // 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 {
      int nMax, nMin;
            for (int i = 0; i < 7; ++i)
                  for (int j = 0; j < 7; ++j) {
                        if (estado.See_Casilla(i, j) == jugador)
                              nMin++;
                        else if (estado.See_Casilla(i, j) != 0)
                              nMax++;
                  }
      return nMax - nMin;
      }
}
```

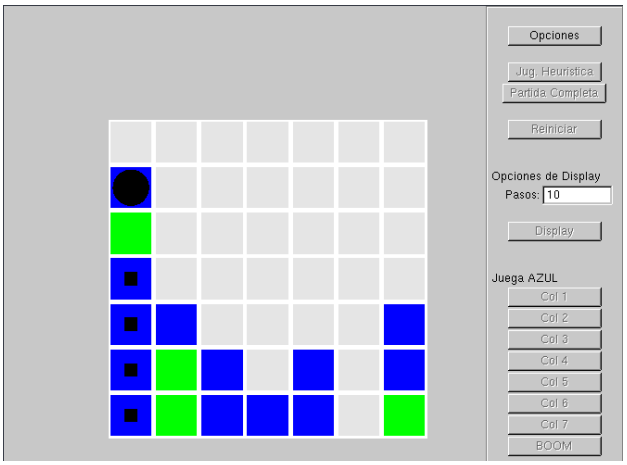
Tras realizar varias partidas de prueba descubrí que una buena estrategia para jugar era intentar destruir con la ficha bomba el mayor número de fichas para que el tablero se fuera llenando cada vez más de fichas del adversario. Por lo tanto la heurística usada es: h = numeroFichasMax – numeroFichasMin

Intentando minimizar nuestro número de fichas y maximizar las del contrario.

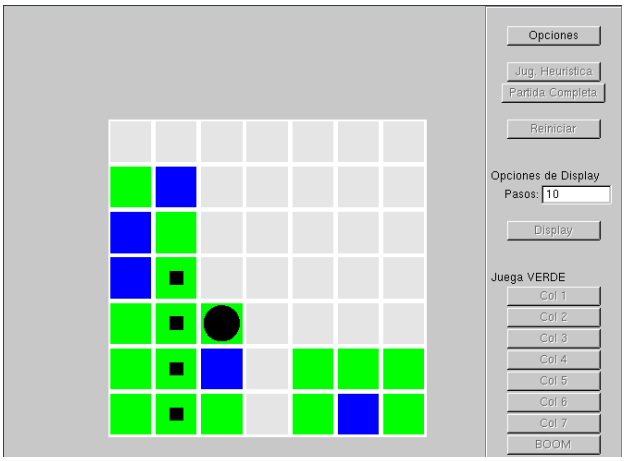
Además se ha implementado una pequeña mejora, cuando se llega al estado de victoria se le resta la profundidad y al llegar al estado de derrota se le suma la profundidad. Esto implica que si encuentra varias maneras de ganar ejecute la que tenga menos movimientos y que si va a perder con todos los movimientos realice la secuencia que tarde más en perder, dando tiempo al adversario de equivocarse.

3. Resultados obtenidos

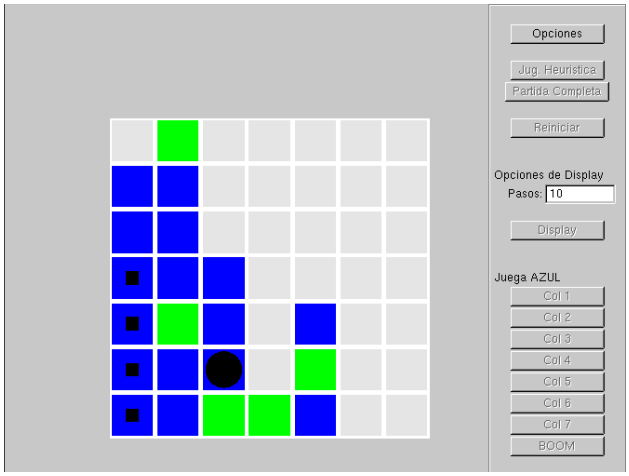
El agente deliberativo implementado vence a los 3 ninjas como jugador 1 y como jugador 2. A continuación hay un ejemplo de los estados finales obtenidos.



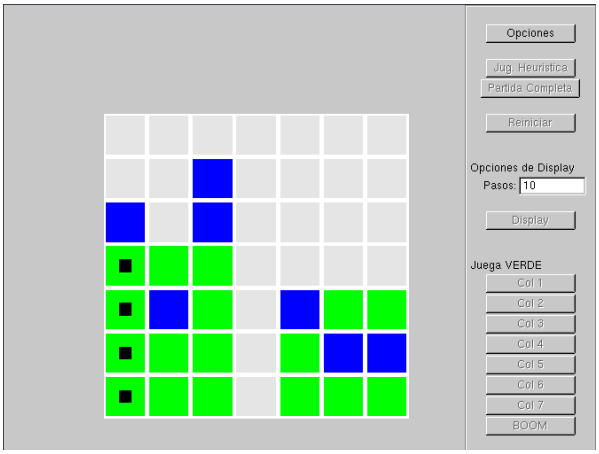
Resultado con Ninja1 como jugador 1



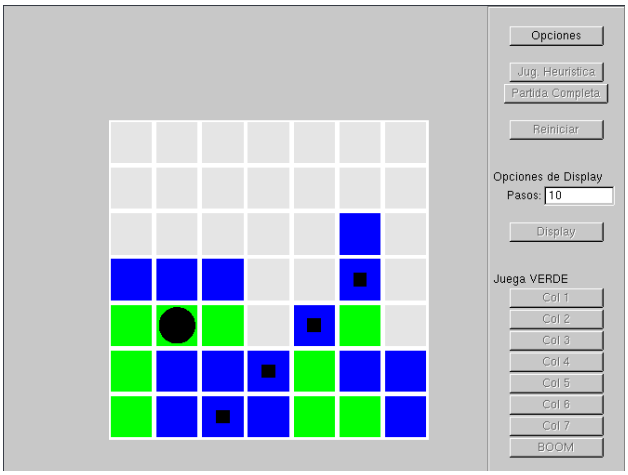
Resultado con Ninja1 como jugador 2



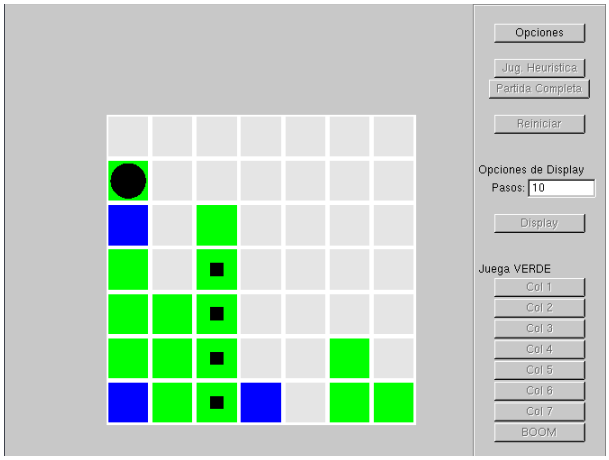
Resultado con Ninja2 como jugador 1



Resultado con Ninja2 como jugador 2



Resultado con Ninja3 como jugador 1



Resultado con Ninja3 como jugador 2