-Análisis del problema-

º Esta práctica consiste en el diseño de algoritmos de búsqueda con adversarios en un entorno de juego. En este caso, el juego es el conocido cuatro en raya o conecta cuatro, pero con una pequeña característica extra (fichas bomba).
Estas fichas bomba añaden al juego una nueva acción, que es la de hacerlas explotar. Al hacerlo, destruiremos todas las casillas existentes en esa fila que sean del adversario, de este modo, todas las casillas posicionadas arriba de estas caerán por gravedad a la posición de las fichas anteriormente explotadas.
Tenemos un objetivo a realizar en esta práctica, que es el de vencer a los tres oponentes (ninja 1, ninja 2 y ninja 3) con nuestra implementación del algoritmo y la heurística. El algoritmo a implementar podía ser o MiniMax o AlphaBeta (consiguiendo en este último una profundidad de búsqueda mayor, y por tanto, mejores resultados).
El principal problema con el que nos encontramos es la limitación en la profundidad de exploración del árbol de juego, debido al elevado coste del algoritmo. Debido a ello, muchas veces o casi todas las veces no llegaremos a nodos terminales que nos den información completa sobre si hemos ganado o perdido, por lo que tendremos que desarrollar una función heurística que evalué los nodos intermedios y decida como de prometedores son.

-Poda Alpha-Beta-

º La implementación de la poda alpha-beta es sencilla, pero para este caso he realizado las siguientes modificaciones para su correcto funcionamiento:

1. La condición de parada será alcanzar la máxima profundidad permitida o que el juego haya terminado.
2. Para distinguir jugador maximizante de minimizante he utilizado el método *JugadorActivo()* para poder distinguir al jugador que invoca la función del otro jugador (el minimizante).
3. La acción ha realizar sólo se genera cuando la profundidad sea 0, es decir, cuando la recursividad se haya resuelto (y claro está, que se cumpla que la valoración sea mayor a alpha).

El resto de la implementación del algoritmo es la usual, ir actualizando beta y alpha según los valores de la evaluación de los nodos, podar si se cumple beta <= alpha etc.
No me detengo en ello pues está en conocimiento de todos su implementación.

-Heurística-

º Pasamos a mi solución para la evaluación de los nodos. Para ello lo he dividido en una función Valoración que contempla 4 casos → Se gana, se pierde, se empata, no se sabe. Es este último caso el que llama a la función “eval” cuya tarea es asignar puntuación según el estado del tablero.

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

   if (ganador == jugador)
      return 99999999.0; // Gana el jugador que pide la valo
   else if (ganador != 0)
      return -99999999.0; // Pierde el jugador que pide la v
   else if (estado.Get_Casillas_Libres() == 0)
      return 0; // Hay un empate global y se ha rellenado co
   else
   {
      int otro = 1;

      if (jugador == 1)
         otro = 2;

      double evalJug = eval(estado, jugador);
      double evalOtro = eval(estado, otro);

      return evalJug - evalOtro;
   }
}
```

Como se puede apreciar en la captura, diferencio entre el jugador y otro (el adversario de turno), calculo las valoraciones de cada uno y las resto (maximización y minimización). Esto nos devuelve el valor más aproximado, según mi criterio, referente al estado del tablero con respecto al jugador principal (nuestra ia). Puede devolver valores negativos (ventaja del adversario) o valores positivos (ventaja nuestra).
Pasemos a la función de evaluación y las heurísticas elegidas.

```
for (double i = 0; i < 7; i++)
{
   for (double j = 0; j < 7; j++)
   {
      if (estado.See_Casilla(i, j) == jugador || estado.See_Casilla(i, j) == jugador + 3)
      {
         if (estado.See_Casilla(i, j + 1) == jugador || estado.See_Casilla(i, j + 1) == jugador + 3)
         {
            cHoriz += 5;
            if (estado.See_Casilla(i, j + 2) == jugador || estado.See_Casilla(i, j + 2) == jugador + 3)
               cHoriz += 15;
         }

         if (estado.See_Casilla(i + 1, j) == jugador || estado.See_Casilla(i + 1, j) == jugador + 3)
         {
            cVert += 5;
            if (estado.See_Casilla(i + 2, j) == jugador || estado.See_Casilla(i + 2, j) == jugador + 3)
               cVert += 15;
         }

         if (estado.See_Casilla(i + 1, j + 1) == jugador || estado.See_Casilla(i + 1, j + 1) == jugador + 3)
         {
            cDiago1 += 5;
            if (estado.See_Casilla(i + 2, j + 2) == jugador || estado.See_Casilla(i + 2, j + 2) == jugador + 3)
               cDiago1 += 15;
         }

         if (estado.See_Casilla(i + 1, j - 1) == jugador || estado.See_Casilla(i + 1, j - 1) == jugador + 3)
         {
            cDiago2 += 5;
            if (estado.See_Casilla(i + 2, j - 2) == jugador || estado.See_Casilla(i + 2, j - 2) == jugador + 3)
               cDiago2 += 15;
         }

         if (estado.See_Casilla(i, j) == jugador && j > 2 && j < 5)
            mid += 1;

         if (estado.See_Casilla(i, j) == jugador + 3)
            cBomba += 25;
      }
```

Los principales objetivos son:

1. Buscar casillas horizontales que formen de 2 a 3 en raya, siendo 2 buena y 3 muy buena. Lo mismo para verticales y diagonales.
2. Se incentiva a la ia a poner ficha en el medio, pues hay más casillas hacia todas las direcciones y por ello, más posibilidades de alcanzar la victoria.
3. También se agregan puntos si hay una ficha bomba en el tablero.
4. Si el adversario tiene como mínimo dos fichas adyacentes en vertical, horizontal o diagonal se premia al bot por bloquear su posible futura jugada (al haber dos turnos seguidos podría ganar con solo dos fichas adyacentes en una jugada ya que completaría con otras dos en la siguiente).

```
      }
      else if (estado.See_Casilla(i, j) == otro || estado.See_Casilla(i, j) == otro + 3)
      {
         if (estado.See_Casilla(i, j + 1) == otro || estado.See_Casilla(i, j + 1) == otro + 3)
         {
            if (estado.See_Casilla(i, j + 2) == jugador)
               cEnem += 20;
         }

         if (estado.See_Casilla(i + 1, j) == otro || estado.See_Casilla(i + 1, j) == otro + 3)
         {
            if (estado.See_Casilla(i + 2, j) == jugador)
               cEnem += 20;
         }

         if (estado.See_Casilla(i + 1, j + 1) == otro || estado.See_Casilla(i + 1, j + 1) == otro + 3)
         {
            if (estado.See_Casilla(i + 2, j + 2) == jugador)
               cEnem += 10;
         }

         if (estado.See_Casilla(i + 1, j - 1) == otro || estado.See_Casilla(i + 1, j - 1) == otro + 3)
         {
            if (estado.See_Casilla(i + 2, j - 2) == jugador)
               cEnem += 10;
         }
      }

      turn 5 * (cVert + cHoriz) + cEspacio + cEnem + cBomba + (cDiago1 + cDiago2) * 0.5;
```

En el punto 1, he tenido en cuenta que fuesen tanto fichas del jugador como fichas bomba del jugador, pues también cuentan a la hora de hacer 4 en raya. No ha sido de igual manera en el punto 4 (bloquear al rival) debido a que nos interesa que las fichas bomba se utilicen para explotar filas que nos den ventaja y no para bloquear. Esto se debe a que surgieron casos en los que mi ia bloqueaba con fichas bomba en verticales y algunas diagonales, la cuales al ir hacia arriba suelen tener toda su fila completamente vacía (no hay potencial de explotar fichas del rival para acomodar el tablero a nuestro jugador).
En cuanto al resultado retornado de la función, le he dado mucho más peso a las verticales y horizontales que al resto de valoraciones. La valoración del resto queda intacta, a excepción de diagonales, que al ser dos (no es necesario tener en cuenta diagonales hacia abajo) he dividido su valoración por la mitad.
La valoración queda así → ***return 5 * (cVert + cHoriz) + cEnem + cBomba + (cDiago1 + cDiago2) * 0.5;***