Práctica Final

Esteban Omelio Puentes Silveira
virtualevan@gmail.com, Xosé Ramón Fernández Oxea #3 1ºG

FORMACIÓN ACADÉMICA
2008-2011 Título de Bachiller (Homologado). IPVCE Máximo Gómez Báez (Camagüey-Cuba)
2011-2013 FP2/ Técnico en Administración de Sistemas Informáticos y Redes
2013-Actualidad Grado de Ingeniería Informática
Actualmente Cursando Tercer año del Grado de Ingeniería Informática

EXPERIENCIA LABORAL
2011 (julio-octubre) JCCE-Camagüey 2
Colaborador auxiliar de administrador de red

2013 Prácticas en ARINFO
Técnico de Adm. de Sistemas Informáticos y Redes.

IDIOMAS
CASTELLANO Lengua Materna
INGLÉS Nivel Medio oral y escrito .
2º Curso escuela de idiomas.

RESUMEN
Juego para tres jugadores que colocarán como máximo 2N reinas y solicitarán jugadas a un jugador configurador que realizará N movimientos (colocará bloques o agujeros) en un tablero de tamaño 4N, ganará el jugador que más reinas coloque y en caso de empate ganan las negras, el bloqueador nunca ganará la partida, pero podrá decidirla aceptando o rechazando las peticiones de movimiento de los otros dos jugadores.

INTRODUCCIÓN
Se ha desarrollado usando Jason Agent Speak. La estrategia a seguir por los jugadores ha sido colocar reinas en las posiciones libres que más casillas amenacen, de esta forma el espacio disponible para colocar es menor y la partida termina más rápidamente, el jugador con las reinas negras seguirá la misma estrategia ya que en cualquier caso depende de las malas elecciones de las blancas y/o del configurador para ganar, las solicitudes de movimientos se realizarán a la segunda posición que amenace más casillas y se seleccionará si es una solicitud de bloque o agujero de forma aleatoria. El configurador esperará a tener una petición de cada uno de los jugadores, entonces evaluará si las casillas solicitadas son libres, si lo son, ejecutará la petición que tenga una posición que amenace más casillas, si solo una de las peticiones fue

realizada a una casilla libre será esta la petición que se ejecutará y si ninguna de las casillas para las que se realizaron peticiones está libre se considerará que no hay una estrategia ganadora y se colocará un bloque en una posición libre del tablero.

CÓDIGO COMPLETO DE LA PRÁCTICA

```
// Agent r0 in project mars.mas2j

/* Initial goals */

!start.

/* Plans */
+!start : playAs(0) <-
  .wait(500);
  !amenazadas;
  !play
  .

+!start : playAs(1).


+!start : not playAs(_) <-
  .wait(500);
  !playConfig
  .

+size(N)<-
  +configMovs(N/4);
  !crearTablero(N).


/* ----- Crea un tablero de casillas libres con el numero de casillas que amenaza cada una ----- */
+!crearTablero(N) <-
        for(.range(X,0,N-1)){
                for(.range(Y,0,N-1)){
                        +free(X,Y,N*N);
                }
        }.

+queen(X,Y) [source(percept)]  <-
        .print("Actualizando base de conocimientos");
        !ocupar(X,Y);
  .wait(1000);
        !amenazadas;
  .wait(1000);
  //.findall(pos(PosX,PosY,R),free(PosX,PosY,R),Lista);
  //.print("OLA K ASE ",Lista);
```

```
  .


/* ----- Comprueba que la casilla esté amenazada ----- */
+!check(X,Y,Check) <-
  if(queen(X,Y)){
    Check = true;
  }
  else {
    +check(false);
    //Filas
    !filaDerecha(X+1,Y,check);
    !filaIzquierda(X-1,Y,check);

    //Columnas
    !columnaSuperior(X,Y-1,check);
    !columnaInferior(X,Y+1,check);

    //Diagonales
    !diagonalSI(X-1,Y-1,check);
    !diagonalSD(X+1,Y-1,check);
    !diagonalII(X-1,Y+1,check);
    !diagonalID(X+1,Y+1,check);
    ?check(Check);
    .abolish(check(_));
  }
  .
-!check(X,Y)<-.print("ERROR CHECK").


/* ----- Elimina casillas que no son libres ----- */
+!ocupar(X,Y) <-
          -free(X,Y,_);
  //Filas
  !filaDerecha(X+1,Y,ocupar);
  !filaIzquierda(X-1,Y,ocupar);

  //Columnas
  !columnaSuperior(X,Y-1,ocupar);
  !columnaInferior(X,Y+1,ocupar);

  //Diagonales
  !diagonalSI(X-1,Y-1,ocupar);
  !diagonalSD(X+1,Y-1,ocupar);
  !diagonalII(X-1,Y+1,ocupar);
  !diagonalID(X+1,Y+1,ocupar).
-!ocupar(X,Y).


/* ----- Actualiza el contador de casillas libres que amenaza cada casilla ----- */
+!amenazadas <-
```

```
            ?size(N);
  +cont(0);
            for(free(X,Y,AM)){
    -+cont(0);
   // .print("ANALIZANDO ", X,",",Y);
    //Filas
    !filaDerecha(X+1,Y,contar);
    !filaIzquierda(X-1,Y,contar);

    //Columnas
    !columnaSuperior(X,Y-1,contar);
    !columnaInferior(X,Y+1,contar);

    //Diagonales
    !diagonalSI(X-1,Y-1,contar);
    !diagonalSD(X+1,Y-1,contar);
    !diagonalII(X-1,Y+1,contar);
    !diagonalID(X+1,Y+1,contar);

    ?cont(Amenazadas);
                -free(X,Y,AM);
    //La casilla X,Y se cuenta como amenazada tanto en filas como columnas como diagonales (-2)
                +free(X,Y,Amenazadas);
   // .print(X,",",Y,",",Amenazadas);
            }
  .abolish(cont(_)).
-!amenazadas<-.print("ERROR AMENAZADAS").

//Filas
+!filaDerecha(X,Y,Z) : size(N) & not block(X,Y) & X<N <-
  if(Z == check & queen(X,Y) & check(false)){
    -+check(true);
  }
  if(Z == ocupar){
    -free(X,Y,_);
  }
  if(Z == contar & free(X,Y,_)){
    ?cont(AUX);
    -+cont(AUX+1);
   // .print(X,",",Y);
  }
  !filaDerecha(X+1,Y,Z).
+!filaDerecha(X,Y,Z).

+!filaIzquierda(X,Y,Z) : size(N) & not block(X,Y) & X>=0 <-
  if(Z == check & queen(X,Y) & check(false)){
    -+check(true);
  }
  if(Z == ocupar){
    -free(X,Y,_);
  }
```

```
  if(Z == contar & free(X,Y,_)){
    ?cont(AUX);
    -+cont(AUX+1);
   // .print(X,",",Y);
  }
  !filaIzquierda(X-1,Y,Z).
+!filaIzquierda(X,Y,Z).

//Columnas
+!columnaSuperior(X,Y,Z) : size(N) & not block(X,Y) & Y>=0 <-
  if(Z == check & queen(X,Y) & check(false)){
    -+check(true);
  }
  if(Z == ocupar){
    -free(X,Y,_);
  }
  if(Z == contar & free(X,Y,_)){
    ?cont(AUX);
    -+cont(AUX+1);
   // .print(X,",",Y,",",Z);
  }
  !columnaSuperior(X,Y-1,Z).
+!columnaSuperior(X,Y,Z).

+!columnaInferior(X,Y,Z) : size(N) & not block(X,Y) & Y<N <-
  if(Z == check & queen(X,Y) & check(false)){
    -+check(true);
  }
  if(Z == ocupar){
    -free(X,Y,_);
  }
  if(Z == contar & free(X,Y,_)){
    ?cont(AUX);
    -+cont(AUX+1);
   // .print(X,",",Y,",",Z);
  }
  !columnaInferior(X,Y+1,Z).
+!columnaInferior(X,Y,Z).

//Diagonales
+!diagonalSI(X,Y,Z) : size(N) & not block(X,Y) & Y>=0 & X>=0 <-
  if(Z == check & queen(X,Y) & check(false)){
    -+check(true);
  }
  if(Z == ocupar){
    -free(X,Y,_);
  }
  if(Z == contar & free(X,Y,_)){
    ?cont(AUX);
    -+cont(AUX+1);
   // .print(X,",",Y);
```

```
  }
  !diagonalSI(X-1,Y-1,Z).
+!diagonalSI(X,Y,Z).

+!diagonalSD(X,Y,Z) : size(N) & not block(X,Y) & Y>=0 & X<N <-
  if(Z == check & queen(X,Y) & check(false)){
    -+check(true);
  }
  if(Z == ocupar){
    -free(X,Y,_);
  }
  if(Z == contar & free(X,Y,_)){
    ?cont(AUX);
    -+cont(AUX+1);
   // .print(X,",",Y);
  }
  !diagonalSD(X+1,Y-1,Z).
+!diagonalSD(X,Y,Z).

+!diagonalII(X,Y,Z) : size(N) & not block(X,Y) & Y<N & X>=0 <-
  if(Z == check & queen(X,Y) & check(false)){
    -+check(true);
  }
  if(Z == ocupar){
    -free(X,Y,_);
  }
  if(Z == contar & free(X,Y,_)){
    ?cont(AUX);
    -+cont(AUX+1);
   // .print(X,",",Y);
  }
  !diagonalII(X-1,Y+1,Z).
+!diagonalII(X,Y,Z).

+!diagonalID(X,Y,Z) : size(N) & not block(X,Y) & Y<N & X<N <-
  if(Z == check & queen(X,Y) & check(false)){
    -+check(true);
  }
  if(Z == ocupar){
    -free(X,Y,_);
  }
  if(Z == contar & free(X,Y,_)){
    ?cont(AUX);
    -+cont(AUX+1);
   // .print(X,",",Y);
  }
  !diagonalID(X+1,Y+1,Z).
+!diagonalID(X,Y,Z).


/* ----- Turnos ----- */
```

```
+player(N) : playAs(N) <- !play.

+player(N) : playAs(M) & not N==M <- .wait(300); .print("No es mi turno.").


/* ----- Jugar ----- */
+!play : configMovs(M) & playAs(P) <-
  if (M > 0){
    .print("Movimientos restantes del configurador: ", M);
    .wait(4900);
    !select(Max1,Max2);
    .print("Maximo: ", Max1);
    !getPosition(Max1, X1,Y1);
    !getPosition(Max2, X2,Y2);
    queen(X1,Y1);
    //El jugador solicita un bloque o un agujero en la segunda posición que amenaza más casillas
    if( .random(N) & N > 0.5){
      .print("El jugador ", P ," solicita un bloque en ", Max2);
      .wait(3000);
      .send(configurer, tell, block(X2,Y2));
    } else {
      .print("El jugador ", P ," solicita un agujero en ", Max2);
      .wait(3000);
      .send(configurer, tell, hole(X2,Y2));
    }
  } else {
    .print("El configurador ha agotado todos sus movimientos, se salta su turno");
    .wait(500);
    !select(Max);
    .print("Maximo: ", Max);
    !getPosition(Max, X,Y);
    queen(X,Y);

  }
  .
-!play <- .print("Juego Finalizado").

+!getPosition(pos(N,X,Y),X,Y).


/* ----- Seleccionar la Posición con mayor número de amenazadas ----- */
+!select(Max) <-
          .wait(700);
  //NumAmenazadas,X,Y
  .findall(pos(N,X,Y),free(X,Y,N),ListaPosiciones);
  .print("Posiciones posibles: ",ListaPosiciones);
  .max(ListaPosiciones,Max).
-!select(Max) <- Max = [].

/* ----- Seleccionar las dos posiciones con mayor número de amenazadas ----- */
```

```
+!select(Max1,Max2) <-
        .wait(700);
  //NumAmenazadas,X,Y
  .findall(pos(N,X,Y),free(X,Y,N),ListaPosiciones);
  .print("Posiciones posibles: ",ListaPosiciones);
  .max(ListaPosiciones,Max1);
  .delete(Max1,ListaPosiciones,NuevaLista);
  .nth(.length(NuevaLista)/5,NuevaLista,Max2).
-!select(Max1,Max2) <- Max1 = [];Max2 = [].


/********************** A partir de aquí se gestiona el configurador ********************************/

/* ----- Turno del configurador ----- */
+!playConfig: configMovs(M) & N > 0 <-
  !amenazadas;
  .wait(1000);
  !putBlock
  .

/* ----- Colocar bloque ----- */
+!putBlock : configMovs(M) & M>0 <-
  .findall(pos(X,Y,Ag), (accepted(block(X,Y,Ag)) | accepted(hole(X,Y,Ag))), Accepted);
  // El configurador escoge cuál de las dos propuestas amenaza más casillas para el momento actual
  // (Impide que se realice una jugada que amenace dicho número de casillas)
  if( .length(Accepted)=2 ){
    .print("Posiciones aceptadas ", Accepted);
    .nth(0,Accepted,pos(I,J,A1));
    .nth(1,Accepted,pos(O,K,A2));
    //Si las dos posiciones seleccionadas están libres selecciona la que más casillas amenace
    if(free(I,J,Amenaz1) & free(O,K,Amenaz2) & Amenaz1>Amenaz2){
      if (accepted(block(I,J,_))){
        block(I,J);
        .print("Bloque colocado en la posición solicitada por el jugador ", A1);
      } else {
        hole(I,J);
        .print("Agujero colocado en la posición solicitada por el jugador ", A1);
      }
    }
    else {
      if(free(I,J,Amenaz1) & free(O,K,Amenaz2) & Amenaz1<Amenaz2){
        if (accepted(block(O,K,_))){
          block(O,K);
          .print("Bloque colocado en la posición solicitada por el jugador ", A2);
        } else {
          hole(O,K);
          .print("Agujero colocado en la posición solicitada por el jugador ", A2);
        }
      } else {
        //Si una de las casillas seleccionas está amenazada y la otra no, selecciona la que no está amenazada
```

```
      if(free(I,J,_) & not free(O,K,_)){
        if (accepted(block(I,J,_))){
          block(I,J);
          .print("Bloque colocado en la posición solicitada por el jugador ", A1);
        } else {
          hole(I,J);
          .print("Agujero colocado en la posición solicitada por el jugador ", A1);
        }
      }
      else {
        if(not free(I,J,_) & free(O,K,_)){
          if (accepted(block(I,J,_))){
            block(O,K);
            .print("Bloque colocado en la posición solicitada por el jugador ", A2);
          } else {
            hole(O,K);
            .print("Agujero colocado en la posición solicitada por el jugador ", A2);
          }
        }
        //Si ambas casillas seleccionadas están amenazadas coloca un bloque en una casilla libre
        else {
          ?free(P,L,_);
          block(P,L);
          .print("Posiciones ganadoras igualadas. Bloque colocado en la primer posición libre");
        }
      }
    }
   }
   .abolish(accepted(_));
  }
  !putBlock;
  .
-!putBlock <- .print("ERROR PUTBLOCK").
+!putBlock <- .print("El configurador no dispone de más movimientos").


/* ------ BLOQUES ------ */
+block(X,Y) [source(percept)] : configMovs(M) <-
  -free(X,Y,_);
  -+configMovs(M-1);
            .print("Actualizando base de conocimientos");
  ?size(Size);
  for(.range(V,0,Size-1)){
    for(.range(W,0,Size-1)){
      if(not free(V,W,_) & not block(V,W) & not hole(V,W)){
        !check(V,W,Check);
        if(Check\==true & not hole(V,W)){
          +free(V,W,0);
          .print("Pos liberada",V,",",W);
        }
      }
```

```
   }
  }
  !amenazadas;
  .wait(1000)
  .

+block(X,Y) [source(Ag)] : not Ag == percept & (accepted(block(_,_,Ag) | accepted(hole(_,_,Ag)))) <-
  -block(X,Y) [source(Ag)];
  .print("Ya se ha aceptado una petición de este agente");
          .send(Ag,tell,decline).

+block(X,Y) [source(Ag)] : not Ag == percept & (not accepted(block(_,_,Ag)) | not
accepted(hole(_,_,Ag))) <-
  -block(X,Y) [source(Ag)];
          .findall(pos(I,J), free(I,J,_), ListaLibres);
          .length(ListaLibres, Num);
          if (Num > 1 & not queen(X,Y) & not hole(X,Y)) {
                  +accepted(block(X,Y,Ag));
                  .send(Ag,tell,accept);
    .print("Aceptado bloque del jugador ", Ag);
          } else {
                  .send(Ag,tell,decline)
          }.


/* ------ AGUJEROS ------ */
+hole(X,Y) [source(percept)] : configMovs(M) <-
  -free(X,Y,_);
  -+configMovs(M-1);
          .print("Actualizando base de conocimientos");
  !amenazadas;
  .wait(1000);
  .

+hole(X,Y) [source(Ag)] : not Ag == percept & (accepted(block(_,_,Ag) | accepted(hole(_,_,Ag)))) <-
  -hole(X,Y) [source(Ag)];
          .print("Ya se ha aceptado una petición de este agente");
          .send(Ag,tell,decline).

+hole(X,Y) [source(Ag)] : not Ag == percept & (not accepted(block(_,_,Ag)) | not
accepted(hole(_,_,Ag))) <-
  -hole(X,Y) [source(Ag)];
          .findall(pos(I,J), free(I,J,_), ListaLibres);
          .length(ListaLibres, Num);
          if (Num > 1 & not queen(X,Y) & not block(X,Y)) {
                  +accepted(hole(X,Y,Ag));
                  .send(Ag,tell,accept);
    .print("Aceptado agujero del jugador ", Ag);
          } else {
                          .send(Ag,tell,decline)
```

}.

EXPLICACIÓN DEL CÓDIGO

Planes

```
+!start : playAs(0) <-
  .wait(500);
  !amenazadas;
  !play
  .
```

Comienza la partida para el turno del jugador 0
Calcula cuántas posiciones amenaza cada casilla libre y luego juega.

```
+!start : playAs(1).
```

Comienza la partida para el turno del jugador 1

```
+!start : not playAs(_) <-
  .wait(500);
  !playConfig
  .
```

Comienza la partida para cuando no es turno de ninguno de los jugadores y juega como configurador.

```
+size(N)<-
  +configMovs(N/4);
  !crearTablero(N).
```

Una vez se conoce el tamaño del tablero se añade a la base de conocimientos el número de movimientos que podrá ejecutar el configurador y se crea un tablero con las dimensiones dadas.

```
+!crearTablero(N) <-
        for(.range(X,0,N-1)){
                for(.range(Y,0,N-1)){
                        +free(X,Y,N*N);
                }
        }.
```

Añade a la base de conocimientos todas las posiciones libres que tendrá el tablero inicialmente, es almacenado de la siguiente forma
free(X,Y,NumAmenazadas)
X indica la columna que ocupará en el tablero
Y indica la fiila que ocupará en el tablero
NumAmenazadas indica el número de posiciones que amenazaría si se colocara una reina en X,Y. (Inicialmente es de N*N, ya que se actualiza con los valores reales antes del primer movimiento)

```
+queen(X,Y) [source(percept)]  <-
        .print("Actualizando base de conocimientos");
        !ocupar(X,Y);
 .wait(1000);
        !amenazadas;
 .wait(1000);
 .
```

En cada uno de los agentes, cuando se recibe una percepción de una reina desde el entorno, se eliminan del tablero las casillas donde se ha colocado la reina y todas las que amenaza, luego se calcula cuántas posiciones amenaza cada casilla libre.

```
+!check(X,Y,Check) <-
 if(queen(X,Y)){
  Check = true;
 }
 else {
  +check(false);
  //Filas
  !filaDerecha(X+1,Y,check);
  !filaIzquierda(X-1,Y,check);
```

```
   //Columnas
   !columnaSuperior(X,Y-1,check);
   !columnaInferior(X,Y+1,check);

   //Diagonales
   !diagonalSI(X-1,Y-1,check);
   !diagonalSD(X+1,Y-1,check);
   !diagonalII(X-1,Y+1,check);
   !diagonalID(X+1,Y+1,check);
   ?check(Check);
   .abolish(check(_));
  }
  .
-!check(X,Y)<-.print("ERROR CHECK").
```

Comprueba si una casilla está amenazada (está amenazada si al realizar un recorrido en las filas, columnas y diagonales en todas las direcciones se encuentra alguna reina antes que un bloque o el final del tablero), en sea caso Check toma valor true; Check se almacena en la base de conocimientos y será true si cualquiera de las exploraciones lo determina así.

```
+!ocupar(X,Y) <-
  .print(X,",",Y);
         -free(X,Y,_);
  //Filas
  !filaDerecha(X+1,Y,ocupar);
  !filaIzquierda(X-1,Y,ocupar);

  //Columnas
  !columnaSuperior(X,Y-1,ocupar);
  !columnaInferior(X,Y+1,ocupar);

  //Diagonales
  !diagonalSI(X-1,Y-1,ocupar);
  !diagonalSD(X+1,Y-1,ocupar);
  !diagonalII(X-1,Y+1,ocupar);
  !diagonalID(X+1,Y+1,ocupar).
-!ocupar(X,Y).
```

Elimina del tablero las casillas amenazas (La posición donde se ha colocado la reina, y las posiciones que estén en la misma fila, columna y diagonales)

```
+!amenazadas <-
        ?size(N);
  +cont(0);
        for(free(X,Y,AM)){
   -+cont(0);
   //Filas
   !filaDerecha(X+1,Y,contar);
   !filaIzquierda(X-1,Y,contar);

   //Columnas
   !columnaSuperior(X,Y-1,contar);
   !columnaInferior(X,Y+1,contar);

   //Diagonales
   !diagonalSI(X-1,Y-1,contar);
   !diagonalSD(X+1,Y-1,contar);
   !diagonalII(X-1,Y+1,contar);
   !diagonalID(X+1,Y+1,contar);

   ?cont(Amenazadas);
   -free(X,Y,AM);
   //La casilla X,Y se cuenta como amenazada tanto en filas como columnas como
diagonales (-2)
   +free(X,Y,Amenazadas);
  }
  .abolish(cont(_)).
 -!amenazadas<-.print("ERROR AMENAZADAS").
```

Actualiza el tablero con el número de casillas que amenaza cada una de las posiciones libres. (free(X,Y,NumAmenazas))

Partiendo de cada una de las posiciones libres se añade una creencia a 0 (cont) y se recorre en cada uno de los sentidos contando y almacenando el número en dicha creencia, que luego se consultará para obtener el número de amenazadas en cada una de las posiciones, entonces se actualiza el valor y se elimina la creencia.

Los siguientes planes se explican juntos ya que cumplen una misma funcionalidad en diferentes "direcciones"

```
//Filas
+!filaDerecha(X,Y,Z) : size(N) & not block(X,Y) & X<N <-
  if(Z == check & queen(X,Y) & check(false)){
    -+check(true);
  }
  if(Z == ocupar){
    -free(X,Y,_);
  }
  if(Z == contar & free(X,Y,_)){
    ?cont(AUX);
    -+cont(AUX+1);
   // .print(X,",",Y);
  }
  !filaDerecha(X+1,Y,Z).
+!filaDerecha(X,Y,Z).

+!filaIzquierda(X,Y,Z) : size(N) & not block(X,Y) & X>=0 <-
  if(Z == check & queen(X,Y) & check(false)){
    -+check(true);
  }
  if(Z == ocupar){
    -free(X,Y,_);
  }
  if(Z == contar & free(X,Y,_)){
    ?cont(AUX);
    -+cont(AUX+1);
   // .print(X,",",Y);
  }
  !filaIzquierda(X-1,Y,Z).
+!filaIzquierda(X,Y,Z).

//Columnas
+!columnaSuperior(X,Y,Z) : size(N) & not block(X,Y) & Y>=0 <-
  if(Z == check & queen(X,Y) & check(false)){
    -+check(true);
```

```
  }
  if(Z == ocupar){
    -free(X,Y,_);
  }
  if(Z == contar & free(X,Y,_)){
    ?cont(AUX);
    -+cont(AUX+1);
   // .print(X,",",Y,",",Z);
  }
  !columnaSuperior(X,Y-1,Z).
+!columnaSuperior(X,Y,Z).

+!columnaInferior(X,Y,Z) : size(N) & not block(X,Y) & Y<N <-
  if(Z == check & queen(X,Y) & check(false)){
    -+check(true);
  }
  if(Z == ocupar){
    -free(X,Y,_);
  }
  if(Z == contar & free(X,Y,_)){
    ?cont(AUX);
    -+cont(AUX+1);
   // .print(X,",",Y,",",Z);
  }
  !columnaInferior(X,Y+1,Z).
+!columnaInferior(X,Y,Z).

//Diagonales
+!diagonalSI(X,Y,Z) : size(N) & not block(X,Y) & Y>=0 & X>=0 <-
  if(Z == check & queen(X,Y) & check(false)){
    -+check(true);
  }
  if(Z == ocupar){
    -free(X,Y,_);
  }
  if(Z == contar & free(X,Y,_)){
    ?cont(AUX);
    -+cont(AUX+1);
   // .print(X,",",Y);
```

```
  }
  !diagonalSI(X-1,Y-1,Z).
+!diagonalSI(X,Y,Z).

+!diagonalSD(X,Y,Z) : size(N) & not block(X,Y) & Y>=0 & X<N <-
  if(Z == check & queen(X,Y) & check(false)){
    -+check(true);
  }
  if(Z == ocupar){
    -free(X,Y,_);
  }
  if(Z == contar & free(X,Y,_)){
    ?cont(AUX);
    -+cont(AUX+1);
   // .print(X,",",Y);
  }
  !diagonalSD(X+1,Y-1,Z).
+!diagonalSD(X,Y,Z).

+!diagonalII(X,Y,Z) : size(N) & not block(X,Y) & Y<N & X>=0 <-
  if(Z == check & queen(X,Y) & check(false)){
    -+check(true);
  }
  if(Z == ocupar){
    -free(X,Y,_);
  }
  if(Z == contar & free(X,Y,_)){
    ?cont(AUX);
    -+cont(AUX+1);
   // .print(X,",",Y);
  }
  !diagonalII(X-1,Y+1,Z).
+!diagonalII(X,Y,Z).

+!diagonalID(X,Y,Z) : size(N) & not block(X,Y) & Y<N & X<N <-
  if(Z == check & queen(X,Y) & check(false)){
    -+check(true);
  }
  if(Z == ocupar){
```

```
   -free(X,Y,_);
  }
  if(Z == contar & free(X,Y,_)){
    ?cont(AUX);
    -+cont(AUX+1);
   // .print(X,",",Y);
  }
  !diagonalID(X+1,Y+1,Z).
+!diagonalID(X,Y,Z).
```

Los parámetros que admiten son X,Y,Z
X indica la columna de la casilla a analizar
Y indica la fila de la casilla a analizar
Z es utilizado para comprobar qué acción se realizará, puede ser ocupar o contar.
En cada uno se comprobará que los valores con los que se llama estén dentro de las dimensiones del tablero

Si Z es igual a "ocupar" se eliminará free(X,Y,_) de la base de conocimientos (la casilla ya no estará libre) y se ejecutará la misma meta recursivamente con la posición siguiente correspondiente a cada caso.
Por ejemplo filaDerecha(X,Y,ocupar) eliminará free X,Y de la base de conocimientos y ejecutará filaDerecha(X+1,Y,ocupar), y así sucesivamente hasta que no se cumpla que X esté dentro de las dimensiones del tablero.
De esta forma se avanza eliminado posiciones desde el punto donde se coloca la reina hasta el fin del tablero en cada uno de los sentidos.

Si Z es igual a "contar" se aumentará en 1 un contador en la base de conocimientos para cada casilla que sea libre por la que se pase. Actúa igual que el ejemplo mostrado anteriormente pero suma las casillas libres que hay en cada una de las direcciones y las va almacenando en una misma creencia, de forma que una vez terminado podamos consultar cuántas casillas quedarían amenazadas si colocáramos una reina en cierta posición.

Si Z es igual a "check" se comprueba si hay una reina en X,Y y que check sea falso, si todo esto se cumple se cambia el valor de Check en la base de conocimientos por true. Actúa igual que el ejemplo mostrado anteriormente pero comprueba si hay reinas antes de bloques o el fin del tablero en cada una de las direcciones.

```
 +player(N) : playAs(N) <- .wait(500); !play.
```

+player(N) : playAs(M) & not N==M <- .wait(300); .print("No es mi turno.").

Se comprueba que cada jugador juegue sólo durante su turno, si no es su turno envía un mensaje y espera.

```
/* ----- Jugar ----- */
+!play : configMovs(M) & playAs(P) <-
  if (M > 0){
    .print("Movimientos restantes del configurador: ", M);
    .wait(4900);
    !select(Max1,Max2);
    .print("Maximo: ", Max1);
    !getPosition(Max1, X1,Y1);
    !getPosition(Max2, X2,Y2);
    queen(X1,Y1);
    //El jugador solicita un bloque o un agujero en la segunda posición que amenaza más
casillas
    if( .random(N) & N > 0.5){
      .print("El jugador ", P ," solicita un bloque en ", Max2);
      .wait(3000);
      .send(configurer, tell, block(X2,Y2));
    } else {
      .print("El jugador ", P ," solicita un agujero en ", Max2);
      .wait(3000);
      .send(configurer, tell, hole(X2,Y2));
    }
  } else {
    .print("El configurador ha agotado todos sus movimientos, se salta su turno");
    .wait(500);
    !select(Max);
    .print("Maximo: ", Max);
    !getPosition(Max, X,Y);
    queen(X,Y);

  }
  .
-!play <- .print("Juego Finalizado").
```

Plan que se ejecuta durante el turno de cada jugador, comprueba si el valor de configMovs en la base de conocimientos es mayor que 0 (si el cunfigurador aún tiene movimientos disponibles), en ese caso espera al menos cinco segundos que es el turno del configurador, luego seseleccionan las dos posiciones libres que amenacen más casillas, se coloca una reina en la primera y tras un tiempo (para actualizar las bases de conocimiento) se solicita al configurador un bloque o un agujero (aleatoriamente) en la segunda. Si el número de movimientos disponibles del configurador no es mayor que 0 se selecciona la posición libre que amenace más casillas y se coloca una reina en ella.

+!getPosition(pos(N,X,Y),X,Y).

Obtiene los valores X e Y a partir de una entrada tipo pos(NumAmenazadas,X,Y)

```
+!select(Max) <-
        .wait(700);
  .findall(pos(N,X,Y),free(X,Y,N),ListaPosiciones);
  .print("Posiciones posibles: ",ListaPosiciones);
  .max(ListaPosiciones,Max).
-!select(Max) <- Max = [].
```

Selecciona el valor la casilla que amenace más posiciones libres:
Busca todas las posiciones libres (free(X,Y,NumAmenazadas)) y las almacena en una lista con la forma (pos(NumAmenazadas,X,Y)). Luego selecciona el valor máximo de dicha lista.

```
+!select(Max1,Max2) <-
        .wait(700);
  //NumAmenazadas,X,Y
  .findall(pos(N,X,Y),free(X,Y,N),ListaPosiciones);
  .print("Posiciones posibles: ",ListaPosiciones);
  .max(ListaPosiciones,Max1);
  .delete(Max1,ListaPosiciones,NuevaLista);
  .nth(.length(NuevaLista)/5,NuevaLista,Max2).
-!select(Max1,Max2) <- Max1 = [];Max2 = [].
```

Selecciona los valores de las casillas que amenacen más posiciones libres:

Busca todas las posiciones libres (free(X,Y,NumAmenazadas)) y las almacena en una lista con la forma (pos(NumAmenazadas,X,Y)). Luego selecciona los dos valores máximos de dicha lista.

```
* ----- Colocar bloque ----- */
+!putBlock : configMovs(M) & M>0 <-
  .findall(pos(X,Y,Ag), (accepted(block(X,Y,Ag)) | accepted(hole(X,Y,Ag))), Accepted);
  // El configurador escoge cuál de las dos propuestas amenaza más casillas para el
momento actual
  // (Impide que se realice una jugada que amenace dicho número de casillas)
  if( .length(Accepted)=2 ){
    .print("Posiciones aceptadas ", Accepted);
    .nth(0,Accepted,pos(I,J,A1));
    .nth(1,Accepted,pos(O,K,A2));
    //Si las dos posiciones seleccionadas están libres selecciona la que más casillas amenace
    if(free(I,J,Amenaz1) & free(O,K,Amenaz2) & Amenaz1>Amenaz2){
      if (accepted(block(I,J,_))){
        block(I,J);
        .print("Bloque colocado en la posición solicitada por el jugador ", A1);
      } else {
        hole(I,J);
        .print("Agujero colocado en la posición solicitada por el jugador ", A1);
      }
    }
    else {
      if(free(I,J,Amenaz1) & free(O,K,Amenaz2) & Amenaz1<Amenaz2){
        if (accepted(block(O,K,_))){
          block(O,K);
          .print("Bloque colocado en la posición solicitada por el jugador ", A2);
        } else {
          hole(O,K);
          .print("Agujero colocado en la posición solicitada por el jugador ", A2);
        }
      } else {
        //Si una de las casillas seleccionas está amenazada y la otra no, selecciona la que no
está amenazada
        if(free(I,J,_) & not free(O,K,_)){
          if (accepted(block(I,J,_))){
            block(I,J);
```

```
          .print("Bloque colocado en la posición solicitada por el jugador ", A1);
        } else {
          hole(I,J);
          .print("Agujero colocado en la posición solicitada por el jugador ", A1);
        }
       }
       else {
        if(not free(I,J,_) & free(O,K,_)){
          if (accepted(block(I,J,_))){
            block(O,K);
            .print("Bloque colocado en la posición solicitada por el jugador ", A2);
          } else {
            hole(O,K);
            .print("Agujero colocado en la posición solicitada por el jugador ", A2);
          }
        }
        //Si ambas casillas seleccionadas están amenazadas coloca un bloque en una casilla
libre
        else {
          ?free(P,L,_);
          block(P,L);
          .print("Posiciones ganadoras igualadas. Bloque colocado en la primer posición
libre");
        }
       }
     }
   }
   .abolish(accepted(_));
  }
  !putBlock;
  .
-!putBlock <- .print("ERROR PUTBLOCK").
+!putBlock <- .print("El configurador no dispone de más movimientos").
```

Plan mediante el cual juega el configurador, se mantiene ejecutándose mientras el configurador disponga de movimientos (comprobando que configMovs en la base de conocimientos tenga un número mayor que 0). Comprueba las propuestas aceptadas por el configurador, si ya se han aceptado dos propuestas de movimientos se procede con la estrategia de selección:

Si las posiciones sugeridas por ambos agentes están libres (no esté amenazada) se realiza el movimiento solicitado en la casilla que más posiciones amenaza (la casilla que más posiciones amenazaría si se colocara una reina en ella).

Si solo una de las posiciones sugeridas está libre (no esté amenazada) será ese el movimiento realizado.

Si ninguna de las posiciones sugeridas está libre (no esté amenazada) se considerará que no hay una estrategia ganadora y se colocará un bloque en una posición libre del tablero.

Si el configurador no dispone de más movimientos simplemente muestr un mensaje que lo indica.

```
/* ------ BLOQUES ------ */
+block(X,Y) [source(percept)] : configMovs(M) <-
 -free(X,Y,_);
 -+configMovs(M-1);
        .print("Actualizando base de conocimientos");
 ?size(Size);
 for(.range(V,0,Size-1)){
  for(.range(W,0,Size-1)){
   if(not free(V,W,_) & not block(V,W) & not hole(V,W)){
    !check(V,W,Check);
    if(Check\==true & not hole(V,W)){
     +free(V,W,0);
     .print("Pos liberada",V,",",W);
    }
   }
  }
 }
 !amenazadas;
 .wait(1000)
 .
```

En cada uno de los agentes, cuando se recibe una percepción de un bloque se elimina del tablero la casilla donde se ha colocado el bloque, se reduce en 1 en la base de conocimientos el número de jugadas disponibles para el configurador y "se libera" se añaden a la lista de casillas

libres las posiciones que con la llegada del nuevo bloque dejan de estar amenazadas, luego se calcula cuántas posiciones amenaza cada casilla libre.

```
  +block(X,Y) [source(Ag)] : not Ag == percept & (accepted(block(_,_,Ag) |
accepted(hole(_,_,Ag)))) <-
    -block(X,Y) [source(Ag)];
    .print("Ya se ha aceptado una petición de este agente");
          .send(Ag,tell,decline).
```

Si se recibe una percepción de un bloque desde un agente del cual ya se ha aceptado una petición anteriormente se rechaza la nueva petición y se muestra un mensaje que lo indica.

```
  +block(X,Y) [source(Ag)] : not Ag == percept & (not accepted(block(_,_,Ag)) | not
accepted(hole(_,_,Ag))) <-
    -block(X,Y) [source(Ag)];
          .findall(pos(I,J), free(I,J,_), ListaLibres);
          .length(ListaLibres, Num);
          if (Num > 1 & not queen(X,Y) & not hole(X,Y)) {
                +accepted(block(X,Y,Ag));
                .send(Ag,tell,accept);
      .print("Aceptado bloque del jugador ", Ag);
          } else {
                .send(Ag,tell,decline)
          }.
```

Si se recibe una percepción de un bloque desde un agente del cuál no se ha aceptado una petición anteriormente se comprueba que hayan posiciones libres en el tablero y la posición solicitada no esté ocupada, si se cumple la comprobación se le envía al agente la aceptación de la petición y se almacena en la base de conocimientos (acepted(block(X,Y,Agente))), si no se cumple la comprobación se rechaza la petición.

```
  /* ------ AGUJEROS ------ */
  +hole(X,Y) [source(percept)] : configMovs(M) <-
    -free(X,Y,_);
    -+configMovs(M-1);
          .print("Actualizando base de conocimientos");
```

```
   !amenazadas;
   .wait(1000);
   .
```

En cada uno de los agentes, cuando se recibe una percepción de un agujero se elimina del tablero la casilla donde se ha colocado el agujero y se reduce en 1 en la base de conocimientos el número de jugadas disponibles para el configurado, luego se calcula cuántas posiciones amenaza cada casilla libre.

```
  +hole(X,Y) [source(Ag)] : not Ag == percept & (accepted(block(_,_,Ag) |
accepted(hole(_,_,Ag)))) <-
   -hole(X,Y) [source(Ag)];
         .print("Ya se ha aceptado una petición de este agente");
         .send(Ag,tell,decline).
```

Si se recibe una percepción de un agujero desde un agente del cual ya se ha aceptado una petición anteriormente se rechaza la nueva petición y se muestra un mensaje que lo indica.

```
  +hole(X,Y) [source(Ag)] : not Ag == percept & (not accepted(block(_,_,Ag)) | not
accepted(hole(_,_,Ag))) <-
   -hole(X,Y) [source(Ag)];
         .findall(pos(I,J), free(I,J,_), ListaLibres);
         .length(ListaLibres, Num);
         if (Num > 1 & not queen(X,Y) & not block(X,Y)) {
                  +accepted(hole(X,Y,Ag));
                  .send(Ag,tell,accept);
     .print("Aceptado agujero del jugador ", Ag);
         } else {
                           .send(Ag,tell,decline)
         }.
```

Si se recibe una percepción de un agujero desde un agente del cuál no se ha aceptado una petición anteriormente se comprueba que hayan posiciones libres en el tablero y la posición solicitada no esté ocupada, si se cumple la comprobación se le envía al agente la aceptación de la petición y se almacena en la base de conocimientos (acepted(hole(X,Y,Agente))), si no se cumple la comprobación se rechaza la petición.

CONCLUSIÓN

Los agentes funcionan correctamente y dentro de los parámetros establecidos para la práctica, no realizan ninguna jugada ilegal y la ejecución completa del código se realiza en un tiempo razonable.

REFERENCES

Manual oficial de la API Jason:
http://jason.sourceforge.net/api/overview-summary.html