PRACTICA FINAL SI NREINAS

###1 OBJETIVOS Modificar las prácticas anteriores del juego de las n reinas de forma que en esta ocasión cada grupo de alumnos tendrá un rol de juego: - Jugador (blancas o negras) : Su objetivo es tratar de ganar el juego colocando más reinas que su rival sin que se ataquen entre si ni sean atacadas. - Configurador : Se encargará de mantener comunicación con los jugadores para colocar bloques o agujeros en el tablero según las indicaciones de estos, no siendo su objetivo ganar la partida. El configurador acuerda con el jugador la posición en la que este quiere colocar el bloque(o agujero) y decide si la posición el la que el jugador quiere colocar la pieza es una estrategia ganadora. Los agujeros solamente impiden que se coloquen reinas en esa posicion. Los bloques además de lo anterior, crean refugios a los ataques de las reinas.

###2 CONDICIONES

Cada grupo de alumnos tendrá asignado un rol de juego:

Jugador (blancas o negras).- Tratará de colocar el mayor número posible de reinas (hasta 2N) en el tablero sin que se ataquen entre sí, ni sean atacadas por otras reinas. **Configurador.- **Su misión será modificar el tablero mediante la colocación de dos tipos de celdas especiales: muros y agujeros

Ambas celdas impiden que se coloque encima una reina.

1_Los muros permiten crear refugios en el ataque de reinas, los agujeros no impiden el ataque. 2_El configurador podrá poner como máximo N piezas nuevas en el tablero entre bloques y agujeros 3_La evaluación de la práctica incluirá una competición entre los programas desarrollados por los distintos grupos. Se desarrollará en un número no inferior a 10 rondas en las que se alternarán los colores y roles de los grupos (blancas siempre empiezan moviendo, despues los configuradores mientras puedan -en esta competición deben respetar su turno de juego, aunque pueden decidir no mover; se entenderá que pasa turno si no ha movido en 5 segundos- y negras cierran el ciclo de movimientos) 4_Cada jugador colocará una pieza por turno.

1_Si el programa no verifica las reglas, se considerará que la práctica está suspensa. 2_Si alguno de los jugadores o el configurador intenta hacer trampas perderá la partida y la práctica se considerará no superada. 3_Los jugadores que juegan con blancas o negras ganan si se dan las mismas circunstancias que en las prácticas anteriores 4_El configurador no puede ganar la partida, solo decidirla. Para ello mantendrá una comunicación constante con los jugadores e intentará colocar sus fichas en las posiciones que acuerde con ellos. 5_El configurador puede llegar a acuerdos con un jugador diferente en cada turno hasta que agote sus movimientos. 6_El configurador llegará a un acuerdo con el jugador que le ofrezca la jugada que permita maximizar una función de beneficio que calcule la razon entre las posiciones ganadoras para negras y las posiciones ganadoras para blancas en la siguiente jugada despues de haber escogido la propuesta de blancas o de negras. Si no consigue determinar que existe una posición ganadora

para ninguna de las porpuestas realizará un movimiento de colocar un bloque en una de las celdas no atacadas del tablero. 7_Una posición es ganadora cuando al escoger esa celda en la siguiente jugada, el jugador que la realizó tiene una estrategia ganadora (existe una combinación de colocación de reinas sin que se añadan más bloques o agujeros) que le permite ganar la partida. 8_Para realizar la comunicación los agentes utilizarán los siguientes mensajes: .send(configurer, tell, block(X,Y)), X es la fila e Y la columna de la posición del tablero en que se quiere poner el bloque .send(configurer, tell, hole(X,Y)), X es la fila e Y la columna de la posición del tablero en que se quiere poner el agujero .send(Ag, tell, accept), Ag = {white, black} .send(Ag, tell, decline), Ag = {white, black}

3 ESTRATEGIAS

3.1. JUGADOR (r1)

3.1.1 ESTRATEGIA COLOCACION

La estrategia seleccionada para resolver el problema se puede dividir en dos partes, en la primera el algoritmo a partir de la posición de las reinas desplegadas en el tablero genera una lista con las coordenadas de todas las posiciones que no estan amenazadas por una reina y se le restan las posicones donde hay agujeros de esta manera partimos de todas las posibles ubicaciones de la siguiente reina a posicionar dejando a la segunda parte del algoritmo la estrategia de colocación, para esta práctica nos hemos decantado por seleccionar una posicion mediante un metodo random, de manera que se selecciona unas coordenadas

aleatorias dentro de las proporcionadas por la lista generada en la primera parte de nuestro algoritmo. Justo antes de colocar la reina comprueba si el agente bloqueador acaba de poner un bloque en la poscicion de manera que no ponga la reina encima del bloque.

3.1.2 ESTRATEGIA COLOCACION BLOQUES

El agente tiene una estrategia de colocación de bloques que se basa en recorrer las posiciones que no estan ocupadas por otra pieza para comprobar si colocando un bloque en esa posiciçon aumentan las casillas no amenazdas con respecto a cuando no estaba el bloque, esta estrategia la implementan tanto el agente como el controlador, de manera que el agente cuando coloca una reina también localiza una posicion para colocar un bloque para transmitirsela al controlador que la analiza y devuelve una respuesta.

3.2. CONFIGURADOR(r0)

El configurador (r0) no tiene como objetivo ganar la partida, por lo que se limita a responder a las peticiones de los jugadores y a decidir en función de las peticiones que recibe de estos si las modificaciones que estos quieren realizar sobre el tablero les ayudan a ganar. En el caso de recibir una petición de hole, el configurador acepta la estrategia solicitada siempre y cuando posicionar un hole en la posición solicitada reduzca el número de posiciones en las que es posible colocar reina. (es decir, no se intenta colocar un hole en una posición no atacada) En caso de recibir una petición de block, el configurador acepta la estrategia solicitada siempre y cuando posicionar un block en la posición solicitada

aumente el número de posiciones en las que es posible colocar reina. (es decir, no se intenta colocar un block en una celda que no libera celdas atacadas)

3.3 MOTIVO DE ELECCIÓN ESTRATEGIAS

Para la primera parte del algoritmo nos decidimos por recopilar todas las posiciones no amenazadas porque nos pareció que nos daba muchas mas posibilidades para la selección de la estrategia de colocación en la segunda parte del algoritmo, como posible ampliación de esta parte se podría ordenar la lista de manera que las coordenadas que mas posiciones no amenazadas amenacen este de primeras con el fin de dificultar al ajente enemigo. Para la segunda parte del algoritmo nos decidimos por seleccionar las coordenadas de manera aleatoria debido a falta de tiempo, pero podría mejorarse implementando diferentes estrategias e incluso utilizar la que mas convenga en cada momento. Con respecto a la colocación de bloques intentamos localizar el bloque optimo de manera que mediante su colocación se optimizasen las posiciones no amenazadas pero debido a falta de tiempo no pudimos testear con seguridad que esto ocurra de manera adecuada. Nos decantamos por que nuestro agente jugador no coloque Holes ya que no tienen ninguna ventaja con respecto a los Bloques de todas maneras el controlador gestiona la correcta colocacion de Holes por si otro agente decide colocarlos.

3.4 PROS Y CONTRAS ESTRATEGIA

3.4.1 PROS

a)La discriminacion de las posiciones no amenazadas da mucho juego para la selección de estrategia de colocación b)Al colocarse en una de estas posiciones te aseguras de que la reina siempre va a estar colocada en una posición viable c)mucho margen para seleccionar la estrategia de colocación al tener todas las posiciones posibles d)mucho margen de amplicación y mejora del algoritmo e)La estrategia de colocación de bloques asegura la optimizacion de posiciones no amenazadas

3.4.2 CONTRAS

a)independientemente de la estrategia de colocación siempre se van a comprobar todas las coordenadas para discriminar las no amenazadas b)no es el algoritmo mas rápido

4 IMPLEMENTACIÓN

La implementación de los agentes se dividen en hechos y reglas en la base de conocimientos y objetivos y tareas en el propio agente en jason

4.1 BASE DE CONOCIMIENTOS PREDICADOS: AMBOS AGENTES

feach([],X,Y,LQ,LB,LL,LL): Recorre una lista pasada como argumento(posiciones libres en el tablero), y a partir de una lista inicial(en nuestro caso vacía) devuelve otra lista con el número de posiciones libres que generaría colocar un bloque en cada posición de las posibles.

```
feach([],X,Y,LQ,LB,LL,LL).
feach([Car|Cdr],X,Y,LQ,LB,ListaInicial,R):-qfor(X,Y,LQ,[Car
|LB],LR)&
                                    longitud(LR,K)&
                                    insertar(K,ListaInicial,LL)
&
                                    feach(Cdr,X,Y,LQ,LB,LL,R).
```

mayorPosicionLista(Contador,[],_,R,R): Compara cada elemento de una lista con un valor de referencia y devuelve la posición del mayor elemento de la lista teniendo en cuenta que como mínimo ha de superar el valor de referencia.

```
    mayorPosicionLista(Contador,[],_,R,R).

    mayorPosicionLista(Contador,[Car|Cdr],Referencia,Posici
onMayor,R):-Car>Referencia&
                                        C=Contador+1&
                                        mayorPosicionLi
sta(C,Cdr,Car,Contador,R).
    mayorPosicionLista(Contador,[Car|Cdr],Referencia,Posici
onMayor,R):-Car<=Referencia&
                                        C=Contador+1&
                                        mayorPosicionLi
sta(C,Cdr,Referencia,PosicionMayor,R).
```

qfor(X,Y,L,R):donde X e Y son el tamaño del tablero cuadrado, es decir si es un 4x4 X sería 0 y Y seria 3, LQ la lista con las posiciones de las reinas en el tablero, LB la lista de bloques y R la lista con las posiciones no amenazadas

```
    qfor(Y,Y,LQ,LB,R):- mirafila(Y,Y,LQ,LB,R,Y).

    qfor(X,Y,LQ,LB,K):- X1=X+1 &
                            concat(MF,R,K)&
                            mirafila(X,Y,LQ,LB,MF,Y)&
                            qfor(X1,Y,LQ,LB,R).
```

concat([A|As],Bs,[A|Cs]): Predicado para concatenar listas.

```
    concat([], Cs, Cs).
    concat([A|As],Bs,[A|Cs]):-
            concat(As, Bs, Cs).
```

mirafila(X,Y,L,R,S):comprueba una fila siendo X la primera posicion normalmente 0 ,Y la posicion a comprobar , LQ la lista de posiciones de reinas,LB la lista de bloques R las posiciones no atacadas a devolver y S el tamaño de tablero

```
    mirafila(X,0,LQ,LB,[],S):-  ataca([X,0],LQ,LB,S).
    mirafila(X,0,_,_,[[X,0]],_).
    mirafila(X,Y,LQ,LB,MF,S):-  Y1=Y-1&
```

```
                                ataca([X,Y],LQ,LB,S)&
                                mirafila(X,Y1,LQ,LB,MF,S).
    mirafila(X,Y,LQ,LB,[[X,Y]|MF],S):-  Y1=Y-1 &
                                        mirafila(X,Y1,LQ,LB,MF,
S).
```

ataca(X,Y,LB,S):comprueba si una posicion X es atacada en horizontal o vertical por la primera reina de la lista Y siendo S el tamaño del tablero necesario para atacaDiag y LB la lista de bloques.

```
    ataca([],[],_,_).
    ataca([X,Z],[[X,Y]|R],LB,S):- mayorque(Z,Y,R1)&
                                  menorque(Z,Y,R2)&
                                  not(hayBloqueVertical(X,R2,
R1,LB)).
    ataca([Z,Y],[[X,Y]|R],LB,S):-mayorque(Z,X,R1)&
                                  menorque(Z,X,R2)&
                                  not(hayBloqueHorizontal(Y,R
2,R1,LB)).
    ataca([Z,Y],[[A,B]|R],LB,S):- Z-A == Y-B&not(hayBloqueD
iagonal([Z,Y],[A,B],LB)).
    ataca([Z,Y],[[A,B]|R],LB,S):- Z-A == -(Y-B)&not(hayBloq
ueDiagonal2([Z,Y],[A,B],LB)).
    ataca(Q,[Car|Cdr],LB,P):-   ataca(Q,Cdr,LB,P).
```

hayBloqueDiagonal(X,Y,Z,T): Comprueba si hay un bloque en una de las diagonales entre la reina y la posicion comprobada

```
    hayBloqueDiagonal([Z,Y],[A,B],[[BX,BY]|R]):-Z-BX == Y-B
Y&
                                                menorque(Z,
A,R1)&
                                                mayorque(Z,
A,R2)&
                                                medio(BX,R1
,R2).
    hayBloqueDiagonal([Z,Y],[A,B],[[_,_]|R]):-hayBloqueDiag
onal([Z,Y],[A,B],R).
```

hayBloqueDiagonal2(X,Y,Z,T): Hace lo mismo que el anterior, con la otra diagonal.

```
    hayBloqueDiagonal2([Z,Y],[A,B],[[BX,BY]|R]):-Z-BX == -(
Y-BY)&
                                                menorque(Z,
A,R1)&
                                                mayorque(Z,
A,R2)&
                                                medio(BX,R1
,R2).
    hayBloqueDiagonal2([Z,Y],[A,B],[[_,_]|R]):-hayBloqueDia
gonal2([Z,Y],[A,B],R).
```

hayBloqueVertical(X,Y,Z,T): Comprueba si hay bloques en la vertical

```
    hayBloqueVertical(X,R1,R2,[[X,K]|_]):-menor(K,R2)&
                                        mayor(K,R1).
    hayBloqueVertical(X,R1,R2,[[_,_]|R]):-hayBloqueVertical
(X,R1,R2,R).
```

hayBloqueHorizontal(X,Y,Z,T): Comprueba si hay bloques en la horizontal

```
    hayBloqueHorizontal(X,R1,R2,[[K,X]|_]):-menor(K,R2)&
                                        mayor(K,R1).
    hayBloqueHorizontal(X,R1,R2,[[_,_]|R]):-hayBloqueHorizo
ntal(X,R1,R2,R).
```

medio(X,Y,Z): Comprueba si el valor de X está en un valor intermedio entre Y y Z

```
    medio(X,Y,Z):-X>Y&X<Z.
```

menor(X,Y): Devuelve true si X es menor que Y

```
    menor(X,Y):-Y>X.
```

mayor(X,Y): Devuelve true si X es mayor que Y

```
    mayor(X,Y):-X>Y.
```

mayorque(X,Y,Z): Devuelve el mayor de los dos elementos

```
    mayorque(X,Y,X):-X>Y.
    mayorque(X,Y,Y):-Y>=X.
```

menorque(X,Y,Z): Devuelve el menor de los dos elementos

```
    menorque(X,Y,X):-X<Y.
    menorque(X,Y,Y):-Y<=X.
```

monta(L,L2):se le pasa una lista de elementos tipo queen(X,Y) y devuelve una lista de pares de objetos [X,Y].

```
    monta([],[]).
    monta([Car|Cdr],[[X,Y]|L]):- despieza(Car,X,Y) &
                                 monta(Cdr,L).
```

despieza(q(X,Y),X,Y):se le pasa un objeto de tiepo queen(X,Y) y devuelve X e Y.

```
    despieza([],[],[]).
    despieza(q(X,Y),X,Y).
```

elemrandom(L,N,E):devuelve E siendo el elemento N de L, en un principio se pretendía conseguir el numero random en prolog y funcionaba pero daba problemas en jason asi que incluimos N que es

generada en Jason (por eso esta comentado rdn)

```
    elemrandom(L,N,E):-//rdn(L,N)&
                       get(L,N,E).
```

get(X,Y,E):devuelve el elemento E que es el numero Y de la lista X.

```
    get([Car|Cdr],0,Car).
    get([Car|Cdr],N,X):- N1=N-1 &
                get(Cdr,N1,X).
```

longitud(L,Lon):devuelve la longitud de la lista L.

```
    longitud([],0).
    longitud([_|T],N):-longitud(T,N0) & N=N0 + 1.
```

parset([X,Y],X,Y):dado un par de elementos devuelve los elementos.

```
    parset([X,Y],X,Y).
```

concat(A, B, C): concatena A y B en C.

```
    concat([], Cs, Cs).
    concat([A|As],Bs,[A|Cs]):-
            concat(As, Bs, Cs).
```

diferencia(A,B,C):diferencia de listas A y B con resultante C

```
    diferencia([],_,[]).
    diferencia([A|B],K,M):- miembro(A,K)& diferencia(B,K,M)
.
    diferencia([A|B],K,[A|M]):- not(miembro(A,K))& diferenc
ia(B,K,M).
```

miembro(X,L):devuelve true si X es miembro de L.

```
    miembro(X,[X|_]).
    miembro(X,[_|Y]):- miembro(X,Y).
```

insertar(X,L,R):insertar un elemento en una lista

```
    insertar(X,[],[X]).
    insertar(X, Lista, [X|Lista]).
```

4.2FUNCIONAMIENTO

En este punto se describirán los planes implementados en JASON para el funcionamiento de los agentes.

4.2.1. CONFIGURADOR(r0)

El configurador se limita a reaccionar a las peticiones de los jugadores, pues su única misión es modificar el tablero con las peticiones de los

Jugadores si dichas peticiones son estrategias que les ayudan a ganar la partida. El configurador reaccionará si recibe alguno de los siguientes percepts: hole(Elx,Ely): Tanto si lo recibe de blancas como de negras. block(Elx,Ely): Tanto si lo recibe de blancas como de negras.

```
    +hole(Elx,Ely)[source(white)]<-
        -hole(Elx,Ely)[source(white)];
        .print("Petición de hole de white recibida, procesa
ndo");
        !tamTablero(Tam);
        .findall(q(C,D),queen(C,D),L);
        ?monta(L,ListaQueens);
        !tablero(X,Y);
        .findall(q(C,D),block(C,D),LB);
        ?monta(LB,ListaBlocks);
        .findall(q(C,D),hole(C,D),LH);
        ?monta(LH,ListaHoles);
        ?qfor(X,Y,ListaQueens,ListaBlocks,ListaLibresAntesH
oles);
        ?diferencia(ListaLibresAntesHoles,ListaHoles,ListaP
osibles);
        ?longitud(ListaPosibles,Longitud);
        if(Longitud<2){
            .send(white,tell,decline);
        }
        else{
            ?diferencia(ListaPosibles,[[Elx,Ely]],ListaAux)
;
```

```
            ?longitud(ListaAux,LongitudAux);
            if(Longitud >  LongitudAux){
                hole(Elx,Ely);
                .send(white,tell,accept);
            }else{
                .send(white,tell,decline);//Si al hacer la
diferencia la longitud de la lista posibles es la misma, la
 posicion en la que quiere colocar no esta disponible. Colo
car un hole ahi seria inutil.
            }
        }.
```

El código en el caso de que source sea black es análogo, con la única diferencia de que los mensajes de respueta se enviarían a black. El configurador comprueba la lista de reinas, la lista de bloques y la lista de holes que hay en el tablero. A partir de ahí, con qfor selecciona la lista de posiciones libres sin tener en cuenta agujeros. Luego con la diferencia entre el resultado de qfor y la lista de holes se obtiene la lista de posiciones disponibles en ese momento. Si solamente queda una posición libre(o ninguna) el configurador rechaza la petición de hole, pues sería contraproducente para el jugador autoprohibirse jugar en su única posición libre. De lo contrario, el configurador hace una diferencia entre la lista de posiciones posibles y la posición en la que se le solicita colocar un hole. Posteriormente calcula la longitud de esa nueva lista. En caso de que la longitud inicial sea mayor que la de la lista con el nuevo hole, eso quiere decir que el hole está cumpliendo su objetivo de prohibir una posición, por lo que el configurador acepta la solicitud y

coloca el hole. De lo contrario, declina la solicitud.

```
    +block(Elx,Ely)[source(white)]<-
        -block(Elx,Ely)[source(white)];
        .print("Petición de block de white recibida, proces
ando");
        //Falta procesarla aquí
        !tamTablero(Tam);
        .findall(q(C,D),queen(C,D),L);
        ?monta(L,ListaQueens);
        !tablero(X,Y);
        .findall(q(C,D),block(C,D),LB);
        ?monta(LB,ListaBlocks);
        .findall(q(C,D),hole(C,D),LH);
        ?monta(LH,ListaHoles);
        ?qfor(X,Y,ListaQueens,ListaBlocks,ListaLibresAntesH
oles);//Posibles sin bloque nuevo
        ?diferencia(ListaLibresAntesHoles,ListaHoles,ListaP
osibles);
        ?longitud(ListaPosibles,LongitudAntes);
        ?concat(ListaBlocks,[[Elx,Ely]],ListaNewBlocks);
        ?qfor(X,Y,ListaQueens,ListaNewBlocks,ListaNewLibres
AntesHoles);//Posibles con bloque nuevo
        ?diferencia(ListaNewLibresAntesHoles,ListaHoles,Lis
taNewPosibles);
        ?longitud(ListaNewPosibles,LongitudDespues);
        if(LongitudDespues<LongitudAntes){
            .send(white,tell,decline);
```

```
        }
        else{
            block(Elx,Ely);
            .send(white,tell,accept);
        }.
```

El código en el caso de que source sea black es análogo una vez más, por lo que se explicará solamente este. Si recibe una petición de bloque(muro) el agente comienza por calcular, como en el caso de los agujeros, la lista de posiciones posibles antes de colocar el bloque, así como su longitud. Posteriormente hace lo propio añadiendo añadiendo el bloque, de forma análoga al caso de los holes. La diferencia con el caso de los holes radica que en esta ocasión lo que se busca al poner un bloque es reducir el número de posiciones posibles en las que se puede jugar, por lo que el agente aceptará la solicitud solo en ese caso.

4.2.2. JUGADOR(r1)

```
    !start.
    +!start: true <-+bandera;
                    +player(0).
    /* Plans */
    +player(N):playAs(N) & bandera<-
                         .print("Empiezo Turno.");
                         !play(N).

    +player(N) : playAs(M) & not N==M <- .wait(300); .print
("Esperando Turno.").
```

En este trozo de código se realiza la gestión de los turnos. Un turno nuevo comienza cuando el percepto player(N) enviado por el entorno coincide con el número de jugador y además el jugador no estaba ya jugando en ese turno. Esto se hace con “bandera”, pues el entorno envía un nuevo percept con el turno cada vez que se coloca reina,bloque o agujero, pero el turno solamente se cambia al colocar reina. Así pues, bandera se utiliza para garantizar que el turno solo comienza si esta existe(ya que se borrará al comienzo de un turno y se volverá a añadir al final de el como se verá más adelante).

```
    +!play(P) <-
        -bandera;
        !tamTablero(Tam);
        .findall(q(C,D),queen(C,D),L);
        ?monta(L,ListaQueens);
        !tablero(X,Y);
        .findall(q(C,D),block(C,D),LB);
        ?monta(LB,ListaBlocks);
        .findall(q(C,D),hole(C,D),LH);
        ?monta(LH,ListaHoles);
        ?qfor(X,Y,ListaQueens,ListaBlocks,ListaLibresAntesH
oles);

        if(.empty(ListaHoles)){?igual(ListaLibresAntesHoles
,ListaPosibles);}
        else{?diferencia(ListaLibresAntesHoles,ListaHoles,L
```

```
istaPosibles);}

        //Montamos lista de posiciones amenazadas y no amen
azadas sin piezas
        ?qfor(X,Y,[],[],ListaTablero);
        ?concat(ListaQueens,ListaBlocks,LInt);
        ?concat(LInt,ListaHoles,ListaPiezas);
        ?diferencia(ListaTablero,ListaPiezas,ListaDiferenci
a);


        //ListaPosibles=Posiciones libre no amenazadas
        //ListaDiferencia=Posiciones amenazadas y no amenaz
adas libres

        //Si hay piezas en todas las posiciones, se acaba e
l juego
        if(.empty(ListaDiferencia)){.print("HE PERDIDO!!");
}
        else{
        if(.empty(ListaPosibles)){
            .print("HE PERDIDO!!");
        }
        else{
            ?longitud(ListaPosibles,Long);
            !aleatorio(Long-1,XNum);
            ?elemrandom(ListaPosibles,XNum,Reina);
            ?parset(Reina,Rx,Ry);
```

```
            //Montamos la base para calcular la pos de bloq
ue
            ?concat(ListaQueens,Reina,ListaNewQueens);
            ?qfor(X,Y,ListaNewQueens,ListaBlocks,ListaNewLi
bresAntesHoles);

            if(.empty(ListaHoles)){?igual(ListaLibresAntesH
oles,ListaNewPosibles);}
            else{?diferencia(ListaNewLibresAntesHoles,Lista
Holes,ListaNewPosibles);}

            ?concat(ListaNewQueens,ListaBlocks,LNewInt);
            ?concat(LNewInt,ListaHoles,ListaNewPiezas);
            ?diferencia(ListaTablero,ListaNewPiezas,ListaNe
wDiferencia);

            ?longitud(ListaNewPosibles,Referencia);
            ?feach(ListaNewDiferencia,X,Y,ListaNewQueens,Li
staBlocks,[],ListaTam);
            ?mayorPosicionLista(0,ListaTam,Referencia,-1,Po
sMayor);
            if(PosMayor == -1){
                if(.empty(ListaNewDiferencia)){
                    .print("HE PERDIDO!!");
                }
```

```
                else{
                    .print("Primer turno");
                    +bandera;
                    queen(Rx,Ry);
                }
            }
            else{
                if(.empty(ListaNewDiferencia)){
                    .print("HE PERDIDO!!");
                }else{
                    ?elemrandom(ListaNewDiferencia,PosMayor
,Bloque);
                    ?parset(Bloque,Bx,By);
                    if(Bx == Rx & By == Ry){
                        +bandera;
                        queen(Rx,Ry);
                    }else{
                        .print("Enviando solicitud de bloqu
e en posicion:",Bx,By);
                        .send(configurer,tell,block(Bx,By))
;
                        .wait(accept[source(configurer)] |
decline[source(configurer)]);
                        .count(accept[source(configurer)],N
accept);
                        .count(decline[source(configurer)],
Ndecline);
                        if(Naccept==1){
```

```
                        -accept[source(configurer)];
                        .print("Aceptada.");
                    }
                    if(Ndecline==1){
                        -decline[source(configurer)];
                        .print("Denegada");
                    }
                    //Asegurando
                    .findall(q(C,D),block(C,D),LFB);
                    ?monta(LFB,ListaFinalBlock);
                    .findall(q(C,D),hole(C,D),LFH);
                    ?monta(LFH,ListaFinalHoles);
                    ?concat(ListaFinalBlock,ListaFinalH
oles,ListaFinal);
                    if(.member([Rx,Ry],ListaFinal)){
                        .print("Fin.");
                    }else{
                        +bandera;
                        queen(Rx,Ry);
                    }
                }
            }
        }
    }.
```

En esta porción de código se observa lo que cada Jugador hace cuando

juega. Al principio el Jugador calcula la lista de posibles posiciones en las que se pueden colocar piezas (llamada ListaPosibles), así como una lista de todas las posiciones del tablero en las que no hay piezas colocadas todavía (ListaDiferencia). Si ListaDiferencia es vacía, eso quiere decir que el tablero está lleno de piezas y por tanto el Jugador con el turno habrá perdido. A continuación el Jugador selecciona de la lista de posiciones posibles un lugar para colocar la reina(sin colocarla todavía). En base al lugar en el que quiere colocar la reina, elige una buena posición para colocar el bloque. Para elegir la posición del bloque se crean ListaNewPosibles y ListaNewDiferencia, teniendo en cuenta que en esta ocasion hay una nueva reina a tener en cuenta(aunqe siga sin estar colocada). La longitud de la ListaNewPosibles será la longitud de referencia para saber si colocar un bloque generará más posiciones libres, la cual será utilizada a la hora de llamar al predicado mayorPosicionLista, para saber si alguna de las longitudes calculadas anteriormente para cada posible posición con el predicado feach es mayor que la longitud de referencia. Al predicado mayorPosicionLista se le pasa como argumento -1, por lo que si dicho argumento no se ha modificado eso quiere decir que no se ha encontrado ninguna posición en la lista anterior que supere el valor de referencia. En caso contrario, se procederá a seleccionar el bloque que está en dicha posición. A continuación se comprueba si la posición de dicho bloque es igual a la posición de la reina seleccionada(por seguridad). A continuación se le envía al configurador la solicitud de colocar el bloque en ese posición, y se espera por su respuesta. Una vez recibida la respuesta, se imprime si esta ha sido aceptada o no. Por último, el Jugador procede a colocar la reina en el tablero en la posición que tenía pensada desde un principio. Antes de colocarla, previamente comprueba una vez más por seguridad

que no está colocando la reina en una posición ya ocupada por un bloque. Esto se debe a que si solamente quedaba esa posición disponible y ya se ha colocado el bloque en ella (en ese mismo turno por ese mismo agente) la partida habrá terminado. Si este último requisito se cumple, el Jugador coloca la reina y crea la bandera, indicando que su turno ha concluído.

4.2.3 COMUNICACION ENTRE AGENTES Y ENTORNO

size(N) PlayAs Player queen block hole

4.3.3 COMUNICACION ENTRE AGENTE Y CONTROLADOR

```
    .send(configurer, tell, block(X,Y)),, X es la fila e Y la columna de la posición del tablero en que se quiere poner el bloque
    .send(configurer, tell, hole(X,Y)),, X es la fila e Y la columna de la posición del tablero en que se quiere poner el agujero
    .send(Ag, tell, accept),, Ag = {white, black}
    .send(Ag, tell, decline),, Ag = {white, black}
```