PRACTICA 2 SI NREINAS

1 OBJETIVOS

Desarrollar un programa (que funcione en Jason) que intente ganar al juego de colocar N reinas sin amenaza en un tablero de tamaño 2N (siendo N variable) contra otro agente programado por otro grupo siendo proporcionado el entorno.

2 CONDICIONES

1_El programa (por turnos) podrá jugar con blancas o con negras.
2_Si el programa juega con blancas gana si es capaz de colocar N reinas en el tablero sin que se amenacen entre sí ni sean amenazadas por las reinas del oponente, o es capaz de colocar un número de reinas mayor que su oponente.
3_Si el programa juega con negras gana si es capaz de colocar en el tablero al menos tantas reinas como su oponente.
4_Cada jugador no puede colocar una reina en una celda que este cubierta por otra reina (ya sea suya o de su adversario) Cada jugador colocará una reina por turno.
5_Si el programa no verifica las reglas, se considerará que la práctica está suspensa.

3 ESTRATEGIA

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 están amenazadas por una reina 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 posición mediante un método random, de manera que se selecciona unas coordenadas aleatorias dentro de las proporcionadas por la lista generada en la primera parte de nuestro algoritmo.

3.1 MOTIVO DE ELECCIÓN ESTRATEGIA

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 agente 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.

3.2 PROS Y CONTRAS ESTRATEGIA

3.2.1PROS

a)La discriminación 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 ampliación y mejora del algoritmo

3.2.1CONTRAS

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 del agente se divide en hechos y reglas en la base de conocimientos y objetivos y tareas en el propio agente en jason

4.1 BASE DE CONOCIMIENTOS

En total en la base de conocimientos hay 21 predicados prolog en los que radica la mayor parte de la lógica de

selección de coordenadas disponibles y colocación de la reina

PREDICADOS

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, L la lista con las posiciones de las reinas en el tablero y R la lista con las posiciones no amenazadas

```
qfor(Y,Y,L,R):- mirafila(Y,Y,L,R,Y).

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

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

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

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

```
ataca([],[],_).
ataca([X,_],[[X,_]|_],_).
ataca([_,Y],[[_,Y]|_],_).

ataca(Q,[Car|Cdr],P):-  ataca(Q,Cdr,P)|
                        atacaDiag(Q,P,[Car|Cdr]).
```

atacaDiag(X,P,Y):comprueba si la posición X es amenazada por la primera reina de la lista Y a una distancia P en alguna de sus diagonales.

```
atacaDiag([X1,X2],P,[[C1,C2]|R]):-  comprueba(X1,X2,P,[[C1,C2]|R])|
                                    nextlvl([X1,X2],P,[[C1,C2]|R]).
```

nextlvl(X,P,Y):aumenta la distancia P para comprobar la totalidad de las diagonales

```
nextlvl([X1,X2],P,[[C1,C2]|R]):-P>0 &
                            P1=P-1 &
                            atacaDiag([X1,X2],P1,[[C1,C2]|R]).
```

comprueba(X1,X2,P,Y):realiza las comprobaciones finales en las diagonales a la distancia P siendo X1 y X2 las coordenadas desglosadas y Y la lista de reinas.

```
comprueba(X1,X2,P,[[C1,C2]|R]):-comprueba1(X1,X2,P,C1,C2)|
                            comprueba2(X1,X2,P,C1,C2)|
                            comprueba3(X1,X2,P,C1,C2)|
                            comprueba4(X1,X2,P,C1,C2)|
                            comprueba5(X1,X2,P,C1,C2)|
                            comprueba(X1,X2,P,R).


comprueba1(X1,X2,P,C1,C2):- X1=C1+P & X2=C2+P.
```

```
comprueba2(X1,X2,P,C1,C2):- X1=C1-P & X2=C2+P.
comprueba3(X1,X2,P,C1,C2):- X1=C1+P & X2=C2-P.
comprueba4(X1,X2,P,C1,C2):- X1=C1-P & X2=C2-P.
comprueba5(X1,X2,_,C1,C2):- X1=C1 & X2=C2.
```

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 tiempo queen(X,Y) y devuelve X e Y.

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

concat(X, Y, R):concatena las listas X e Y en R.

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

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).
```

checkTurno(Jugador,Turno,X):comprueba si es el turno del jugador X.

```
checkTurno(Jugador,Turno,1):- Jugador = Turno.
checkTurno(Jugador,Turno,0):- not(Jugador=Turno).
```

turno([Car|Cdr],Car):devuelve el primer elemento de una lista.

```
turno([Car|Cdr],Car).
```

4.2 IMPLEMENTACIÓN AGENTE

En este punto se describirán los planes implementados en JASON para el funcionamiento del agente.

4.2.1FUNCIONAMIENTO

```
+!start:true <
    Jugador=0;
    !jugadas(Njugadas);
    !tamTablero(Tam);
    !partida(Njugadas,Jugador,Tam).
```

El funcionamiento del agente sigue los siguientes pasos:
1_Asigna a Jugador el número de jugador, que puede ser 0 o 1. El jugador 0 juega con
blancas (empieza moviendo), mientras que el jugador 1 juega con negras(debe esperar movimiento del otro jugador).
2_Obtiene el número máximo de jugadas a realizar (que dependerá del tamaño del tablero) y lo almacena en Njugadas.
3_Obtiene el tamaño del tablero y lo almacena en Tam.
4_Inicia la partida mediante !partida(Njugadas,Jugador,Tam).

```
+!jugadas(Njugadas):size(N)<-
                    Njugadas=N/2.
```

Obtiene el número de jugadas a realizar. Para ello se recoge el tamaño del tablero enviado por el entorno en size(N) y se divide entre 2. Dado que habrá dos jugadores, si el tablero tiene tamaño 8, cada jugador dispondrá de 4 movimientos.

```
+!tamTablero(Tam):size(N)< -
                    Tam=N.
```

Obtiene el tamaño del tablero enviado por el entorno en size(N) y se le asigna a Tam dicho valor.

```
+!partida(Njugadas,Jugador,Tam):Njugadas>0<-
        if(Jugador == 0 & Njugadas == Tam/2){//Si es el primer turno con blancas
            .print("Jugador:",Jugador,"Jugadas restantes:",Njugadas);
            !jugar(1);
            !partida(Njugadas 1,Jugador,Tam);
        };
        if(Jugador == 1 & Njugadas == Tam/2){//Si es el primer turno con negras
            .wait(1000);
            !jugar(0,Jugador); //Espera turno
        };
        !jugar(0,Jugador);//Espera turno
        .print("Jugador:",Jugador,"Jugadas restantes:",Njugadas);
        !jugar(1);
        !partida(Njugadas 1,Jugador,Tam).
+!partida(0,Jugador,Tam)< -
        .print("Jugador:",Jugador," termina la partida.");
        .kill_agent(r0).
```

El código de !partida funciona de manera recursiva. Cuando se hace la primera llamada desde !start, a partida se le envía el número de jugadas totales en Njugadas, que será la mitad del tamaño del tablero almacenado en Tam.

Así pues, si es la primera vez que el agente llama a !partida, deberá actuar de una manera o de otra en función de si es el Jugador0 o el Jugador1.

En caso de que sea el Jugador0, realizará la primera jugada llamando a !jugar(1) y después realizará una nueva llamada recursiva a !partida, teniendo en cuenta que en esta ocasión el número de jugadas se reducirá en una(pues ya ha jugado una vez). La siguiente vez que comience el ciclo, el agente llamará a !jugar(0,Jugador) (pues no entra en ninguno de los dos if ya que número de jugadas restante ya no coincide con la mitad del tamaño del tablero). Con esa llamada el agente esperará a que se le asigne su turno (se explicará mas adelante). Una vez se le asigne el turno jugará y volverá a hacer la llamada recursiva. Esto será así hasta que Njugadas tenga el valor 0, en cuyo caso se llamará a !partida(0,Jugador,Tam) para terminar la partida.

En caso de que sea el Jugador1, el funcionamiento es similar, con la única diferencia de que la primera vez que se llama a !partida, en lugar de jugar como en el caso del Jugador0, deberá esperar a que dicho jugador haga su movimiento. En este punto se ha introducido un .wait(1000) para evitar problemas con el entorno, pues el entorno no asigna un turno a un jugador hasta que se haga la primera jugada. Es por ello que se utiliza el wait para garantizar que cuando el agente entre en la espera de turno, este turno ya tendrá un valor proporcionado por el entorno. A partir de dicho momento el funcionamiento es idéntico al explicado para el Jugador0.

```
+!jugar(1)< -
    .findall(q(C,D),queen(C,D),L);
    ?monta(L,Lista);
    !tablero(X,Y);
    ?qfor(X,Y,Lista,ListaPosibles);
    .print("Sin Limpiar: ",ListaPosibles);
    ?longitud(ListaPosibles,Long);
    !aleatorio(Long 1,Num);
    ?elemrandom(ListaPosibles,Num,Ele);
    ?parset(Ele,Elx,Ely);
    move_towards(Elx,Ely);
    put(queen).
```

Con !jugar(1) se le indica al agente que es su turno de jugar reina. Para ello:
1_Localiza con .findall() todas las reinas que hay colocadas actualmente en el tablero (información que le envía el entorno).

2_Llama al predicado de la base de conocimientos ?monta, obteniendo un listado de las posiciones ocupadas.

3_Llama a !tablero(X,Y), lo cual le proporciona un rango de valores en función del tamaño del tablero necesario para el funcionamiento del predicado de la base de conocimientos qfor.

4_Tras la llamada a qfor obtiene una lista con las posiciones en las que es posible colocar reina.

5_ Las siguientes llamadas tienen la finalidad de escoger un elemento aleatorio de la anterior lista de posibilidades. Una vez seleccionado, se llama a move_towards (función del entorno) para mover el agente a la posición indicada y put(queen) para pintar una reina en dicha posición.

```
+!tablero(X,Y):size(N)< -
    X=0;
    Y=N 1.
```

A partir del tamaño del tablero, devuelve dos valores X e Y, siendo X=0 e Y el tamaño del tablero menos 1.

```
+!aleatorio(Long,Num)< -
    .random(X);
    Y = X*Long;
    Num= math.round(Y).
```

Obtiene un número aleatorio cuyo valor máximo será la longitud indicada por Long.

```
+!jugar(0,Jugador)<
    .findall(N,player(N),L);
    ?turno(L,Turno);
    ?checkTurno(Jugador,Turno,T);
    if(T==0){
        !jugar(0,Jugador);
    };
    if(T==1){
    }.
```

En el código de !jugar(0,Jugador) se implementa la espera de turno. Para ello se recoge el turno proporcionado por el

entorno con .findall(), lo cual devuelve una lista con el elemento player(X).

Luego se llama al predicado de la base de conocimientos turno para obtener el valor X, que identifica al jugador que tiene que mover a continuación.

A continuación se llama al predicado de la base de conocimientos checkTurno, que devuelve 1 en caso de que Jugador=Turno, y 0 en caso contrario.

En caso de que devuelva un 0, se volverá a llamar a la espera recursivamente, por lo que el agente permanecerá en espera hasta que le toque mover pieza.

En caso de que devuelva un 1, no hace nada y el agente vuelve a donde hizo la llamada a la espera de turno (es decir, a !partida, haciendo la siguiente instrucción que se le indica secuencialmente, que es jugar(1)).

4.2.3 COMENTARIOS SOBRE LA IMPLEMENTACIÓN DEL AGENTE

1) La prueba del funcionamiento del agente se ha realizado asignando el mismo código a dos agentes r0 y r1 (con valores de Jugador 0 y 1 respectivamente).

Con dicha prueba se detectó un problema después del tercer turno, en el cual el agente que empezaba a jugar, r0, jugaba todos los turnos a partir de ese. Tras estudiar el problema, se detectó que esto era debido a que cada vez que se hacía una llamada put(queen) desde un agente al entorno, el entorno añadía un nuevo percept player(Turno). Sin embargo, el percept anterior no era borrado, por lo que el agente disponía en ese momento de player(0) y player(1) simultáneamente, otorgando siempre el turno al agente 0 por ello.

No hemos sabido si la solución a dicho problema era posible desde el código del agente o era un problema con el entorno, por lo que la final hemos optado por añadir a la función putQueen() del entorno las líneas de código para borrar el percept del turno anterior antes de asignar el siguiente.

Esto se corresponde con:

```
Literal turnx = Literal.parseLiteral("player(" + queensPlaced%2 + ")");
removePercept(turnx);
```

De esta forma conseguimos solucionar el problema anterior y los agentes jugaban sus turnos correctamente.

2) Para todo el desarrollo de la práctica, hemos asumido que las coordenadas de la posición de la reina (X,Y) identifican:

X : Eje horizontal (es decir, la columna)

Y: Eje vertical (es decir, la fila)

5 CÓDIGO COMPLETO

```
/* Initial beliefs */

qfor(Y,Y,L,R):- mirafila(Y,Y,L,R,Y).

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

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

```
ataca([],[],_).
ataca([X,_],[[X,_]|_],_).
ataca([_,Y],[[_,Y]|_],_).

ataca(Q,[Car|Cdr],P):-  ataca(Q,Cdr,P)|
                        atacaDiag(Q,P,[Car|Cdr]).

atacaDiag([X1,X2],P,[[C1,C2]|R]):-  comprueba(X1,X2,P,[[C1,C2]|R])|
                                    nextlvl([X1,X2],P,[[C1,C2]|R]).

nextlvl([X1,X2],P,[[C1,C2]|R]):-P>0 &
                                P1=P-1 &
                                atacaDiag([X1,X2],P1,[[C1,C2]|R]).



comprueba(X1,X2,P,[[C1,C2]|R]):-comprueba1(X1,X2,P,C1,C2)|
                                comprueba2(X1,X2,P,C1,C2)|
                                comprueba3(X1,X2,P,C1,C2)|
                                comprueba4(X1,X2,P,C1,C2)|
                                comprueba5(X1,X2,P,C1,C2)|
                                comprueba(X1,X2,P,R).


comprueba1(X1,X2,P,C1,C2):- X1=C1+P & X2=C2+P.
comprueba2(X1,X2,P,C1,C2):- X1=C1-P & X2=C2+P.
comprueba3(X1,X2,P,C1,C2):- X1=C1+P & X2=C2-P.
comprueba4(X1,X2,P,C1,C2):- X1=C1-P & X2=C2-P.
comprueba5(X1,X2,_,C1,C2):- X1=C1 & X2=C2.

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

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

concat([], Cs, Cs).

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

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


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

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

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

checkTurno(Jugador,Turno,1):- Jugador = Turno.
checkTurno(Jugador,Turno,0):- not(Jugador=Turno).

turno([Car|Cdr],Car).
```

```
/* Initial goal */
!start.
/* Plans */

+!start:true <-
    Jugador=0;//Selecci?n de jugador 0 o 1
    !jugadas(Njugadas);
    !tamTablero(Tam);
    !partida(Njugadas,Jugador,Tam).

+!partida(Njugadas,Jugador,Tam):Njugadas>0<-
        if(Jugador == 0 & Njugadas == Tam/2){//Si es el primer turno con blancas
            .print("Jugador:",Jugador,"Jugadas restantes:",Njugadas);
            !jugar(1);
            !partida(Njugadas-1,Jugador,Tam);
        };
        if(Jugador == 1 & Njugadas == Tam/2){//Si es el primer turno con negras
            .wait(1000);
            !jugar(0,Jugador); //Espera turno
        };
            !jugar(0,Jugador);//Espera turno
            .print("Jugador:",Jugador,"Jugadas restantes:",Njugadas);
            !jugar(1);
            !partida(Njugadas-1,Jugador,Tam).

+!partida(0,Jugador,Tam)<-
.print("Jugador:",Jugador," termina la partida.");
.kill_agent(r0).



+!jugar(1)<-
        .findall(q(C,D),queen(C,D),L);
        ?monta(L,Lista);
        !tablero(X,Y);
        ?qfor(X,Y,Lista,ListaPosibles);
        .print("Sin Limpiar: ",ListaPosibles);
        ?longitud(ListaPosibles,Long);
        !aleatorio(Long-1,Num);
        ?elemrandom(ListaPosibles,Num,Ele);
        ?parset(Ele,Elx,Ely);
        move_towards(Elx,Ely);
        put(queen).

+!jugar(0,Jugador)<-
    //!turnoActual(Turno);
    .findall(N,player(N),L);
    ?turno(L,Turno);
    ?checkTurno(Jugador,Turno,T);
    if(T==0){
        !jugar(0,Jugador);//sigue esperando
    };
    if(T==1){
        //Sale de aqu?
    }.
```

```
//+!turnoActual(Turno):player(Np)<-
//Turno=Np.

+!jugadas(Njugadas):size(N)<-
Njugadas=N/2.

+!aleatorio(Long,Num)<-
.random(X);
Y = X*Long;
Num= math.round(Y).


+!tablero(X,Y):size(N)<-
    X=0;
    Y=N-1.

+!tamTablero(Tam):size(N)<-
    Tam=N.
```