N-Queens. (Competición)

> Daniel Camba Lamas.
>
> Diego Casanova José.
>
> Román Puga Quintairos.

Se trata de 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), bajo las siguientes condiciones:

- *El programa (por turnos) podrá jugar con blancas o con negras.*
- *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.*
- *Si el programa juega con negras gana si es capaz de colocar en el tablero al menos tantas reinas como su oponente.*
- *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.*

Para ello se ha seguido la siguiente estrategia, mediante la a continuación detallada implmentación.

1) Estrategia

En primer lugar se obtiene de la **BB** todos los *percepts* de `queen(X,Y)` haciendo con ellos una lista a partir de la cual se calculan las *posiciones libres*, es decir que no están amenazadas en base a las reinas colocadas actualmente.

En base a la lista de *posiciones libres* se escoge una posición al azar, se mueve al agente hasta dicha posición y colocamos una reina.

Motivación

Recopilar la totalidad de casillas libres, daba un abanico de opciones mucho mayor que una comprobación lineal o similares, además que podría permitir la inclusión de estadísticas para la posterior selección de la posición en la que posicionar la reina. En esta ocasión, dudamos entre escoger la última posición de la lista ó una aleatoria, finalmente se ha utilizado la selección aleatoria ya que aunque resulta menos controlable para nosotros, también es menos predecible para el enemigo.

2) Implementación

La implementación del agente se divide en hechos y reglas en la base de conocimientos y objketivos y tareas en elpropio agente en jason En nuestra base de conocimientos hay 9 predicados prolog que se encargan de la obtención de las casillas no amenzadas.

Diferencias entre `r0.asl` y `r1.asl`

En R0 contamos con la presencia del plan `+!start` ya que será R0 el que empiece el juego, de forma que lanzaremos a nuestro jugador 0 por ser el primer turno.

```
1  /* JASON */
2  !start.
3  +!start: true <- +player(0).
4  +player(0) <- !do(0); -player(0)[source(percept)]; .wait({ -
   player(1)[source(percept)] }).
5
6  +!do(P) <-
```

En R1 como en R0 la única intención recogida es el *percept* de player(N) donde para R0 N=0 y para R1 N=1. Aquí lo que haremos será lanzar el plan `!do(N)` que será donde colocaremos la reina en base al estado del tablero, a continuación eliminaremos nuestro *percept* y aguardaremos hasta el final del turno rival.

```
/* JASON */
+player(1) <- !do(1); -player(1)[source(percept)]; .wait({ -
player(0)[source(percept)] }).

+!do(P) <-
```

Predicados

iterator(X,Y,L,R): X e Y nos dan el tamaño del tablero, L el estado actual del tablero y R las casillas no amenazadas.

link(X,Y,R): Un simple metodo para concatenar dos listas en una *R*esultante.

lookfor(X,Y,L,R,S): Realiza la comprobación de una hilera de casiilas, X la primera posicion normalmente 0, Y será la posicion a comprobar , L el estado del tablero, R las casillas no atacadas y S el tamaño de tablero.

threat(X,L,S): X es la posición a comprobar, L las lista con la que tenemos que comparar y de la cual cogemos la primera reina, S es el tamaño del tablero. Comprobamos verticales, horizonatales y, diagonales mediante la llamada a `checkloop` que nos realiza la comprobación de la totalidad de las diagonales, mediante `check` para comprobar la diagonal a una distancia P y `onemore` para aumentar dicha P, osea para aumentar el alcance.

parser([[X,Y]|_],X,Y): Nos permite coger las coordenadas de la primera posición de una lista.

```
/* PROLOG */

iterator(Y,Y,L,R):- lookfor(Y,Y,L,R,Y).
iterator(X,Y,L,K):- X1=X+1 & link(MF,R,K) &
lookfor(X,Y,L,MF,Y) & iterator(X1,Y,L,R).

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

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

threat([],[],_).
threat([X,_],[[X,_]|_],_).
threat([_,Y],[[_,Y]|_],_).
threat(Q,[Car|Cdr],P):- threat(Q,Cdr,P) | checkloop(Q,P,
[Car|Cdr]).

checkloop([X1,X2],P,[[C1,C2]|R]):-
check(X1,X2,P,[[C1,C2]|R])|onemore([X1,X2],P,[[C1,C2]|R]).

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

check(X1,X2,P,[[C1,C2]|R]):- X1=C1+P & X2=C2+P | X1=C1-P &
X2=C2+P | X1=C1+P & X2=C2-P | X1=C1-P & X2=C2-P | X1==C1 &
X2==C2.

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

Plan (`+!do(P)`)

```
+!do(P) <-
  .print("Jugador: ", P); //Jugador P =[0|1].
  ?size(ES); S=ES-1; //Si el tablero es de 8, ES=8, S=7.
  .findall([A,O],queen(A,O),BB); //Obtención del estado actual
del tablero.
  .print("Tablero actual: ", BB); //Lo mostramos.
  ?iterator(0,S,BB,FP); //Obtención de casillas libres
mediante los predicados.
  .print("Posiciones libres: ", FP); //Las mostramos.
  if(not(.empty(FP))) //Si ha casillas libres.
  {
      .shuffle(FP,SFP); //Reordenamos de manera aleatoria.
      ?parser(SFP,X,Y); //Cogemos ahora la primera posición.
      move_towards(X,Y); //Nos movemos a dicha posición.
      put(queen); //Colocamos una reina.
  }
  else //Si no hay más casillas disponibles.
  {
    .print("FIN."); //Decimos que se acabó.
    .kill_agent(r1); //Eliminamos el agente rival.
    .kill_agent(r0); //Eliminamos nuestro agente.
  }.
```

• Comentarios

Se ha añadido un Entorno alternativo denominado `AltEnv` con el que nos ha sido más sencillo visualizar el estado del tablero, sabemos que no tiene relevancia pero, simplemente nos resultó más práctico.