Juan García Santos alu01010325583

Juego de las damas en Prolog
Informe detallado sobre el desarrollo

Introducción

Como proyecto final de prolog, decidí implementar el juego de las damas en prolog. Como jugador, podrás seleccionar quien comienza, si tú, o una IA implementada empleando el algoritmo minimax (con profundidad limitada). Para lograrlo, me han servido de inspiración artículos y videos sobre este tema, pero siempre manteniendo un desarrollo original.

Planteamiento Inicial

Hechos

Sabemos que se trata de un juego de 1 contra 1, que cada jugador está representado por un colo (blanco o negro), que hay dos tipos de pieza (dama y rey) y que solo se puede pasar de dama a rey, no a la inversa. Todo esto lo planteé como los hechos base del juego:

```
%-----Hechos-----%

%%%%%%%%%%%%%%%%%

esDama(pb). %pieza blanca o negra

esDama(pn).


esRey(rb). %rey blanco o negro

esRey(rn).


esBlanca(pb). %pieza dama o rey blancos

esBlanca(rb).


esNegra(pn). %pieza dama o rey negros

esNegra(rn).


esVacia(va). %casillavacia


esPieza(X) :- esDama(X) ; esRey(X). %si es dama o rey, se considera pieza


legal(negro, pn). % es legal que el jugador negro mueva el rey o las piezas negrs
```

```
legal(negro, rn).


legal(blanco, pb).% es legal que el jugador blanco mueva el rey o las piezas
blancas

legal(blanco, rb).


%las blancas son opuestas a las negras y viceversa

piezaOpuesta(X,Y):- esBlanca(X), esNegra(Y).

piezaOpuesta(X,Y):- esNegra(X), esBlanca(Y).


%la piezas de cada color promocionan a rey del mismo color

promocionar(pb, rb).

promocionar(pn, rn).


%color en cada turno

siguienteJugador(blanco, negro).

siguienteJugador(negro, blanco).


%jugador en cada turno

siguienteTurno(minimax, humano).

siguienteTurno(humano, minimax).


% obtiene el rango de accion de una pieza, asi distinguimos por alcance entre una
dama y un rey
```

```
rangoPieza(Pieza, Rango) :- esRey(Pieza), Rango is 8; esDama(Pieza), Rango is 1.


%conversion de valores al output Gráfico
simbolo(nl, ' _ ').
simbolo(va, ' _ ').
simbolo(rn, ' N ').
simbolo(rb, ' B ').
simbolo(pn, ' n ').
simbolo(pb, ' b ').
```

Tras esto, decidí definir las estructuras y formas de acceso a la información que codificarían lo que es la lógica base del juego. Escogí emplear átomos como una especie de variables globales que me permitieran mantener seguimiento del curso del juego (Jugador actual, tablero, piezas...), y también que el tablero sería una lista de casillas, cada una con un valor asignado. Así mismo, decidí emplear un sistema de coordenadas X, Y respecto del tablero de 8x8 y pensé en un conversor de coordenadas para poder obtener la posición respecto a la lista Tablero de cualquier elemento. Además, para comenzar a hacer pruebas, definí un predicado que inicializará un tablero con las piezas colocadas, y otro para leer por consola que jugador deseas ser. Hay que comentar, que en esta bloque de código se aprecian dos átomos más (estado y simulation), además de predicados que permiten modificar o acceder sus valores; esto es porque más adelante me di cuenta de que necesitaría almacenar dos tipos más de tablero, uno temporal en el que el algoritmo pueda probar los movimientos que desee y otro que establezca el estado en cada momento, ya sea en estado de simulación o en estado de tablero, con las fichas colocadas en la misma posición que al acabar el turno anterior.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%---Getters y Setters---%

%%%%%%%%%%%%%%%%%%%%%%%%

% getter de la pieza en las coordinadas X, Y del tablero

getPieza(X, Y, Pieza) :- convertCoord(X, Y, Pos), getTablero(Tablero), nth0(Pos, Tablero, Pieza).


%Como el getter de encima, pero emplea la coordenada ya convertida en un numero de dos dígitos

findPieza(PosPieza, Pieza) :- between(0, 63, PosPieza), getTablero(Tablero), nth0(PosPieza, Tablero, Pieza).


% getter y setter del estado actual del tablero

getEstado(Estado) :- b_getval(estado, Estado). %unifica el valor del átomo estado con la variable Estado

setEstado(Estado) :- b_setval(estado, Estado). %asocia el valor de Estado al átomo estado


%getters y setters del jugador

leerJugador(Jugador, Number):- nl, write('Elija el Jugador '), write(Number), write(" : "), nl, printJugadores, read(Jugador), between(0, 1, Jugador).

leerJugador(Jugador, Level):- nl, write('Ese Jugador no existe.'), nl, leerJugador(Jugador, Level).


setJugador(JugadorNumber, 0) :- b_setval(JugadorNumber, humano).

setJugador(JugadorNumber, 1) :- b_setval(JugadorNumber, minimax).
```

```
%getters del tablero actual

getTablero(Tablero):- getEstado(Estado), %obtiene el estado actual

                      getTablero(Tablero, Estado). %y devuelve el tablero
asociado al estado actual

getTablero(Tablero, tablero):- b_getval(tablero, Tablero). %devuelve en tablero
aquel asociado al átomo tablero

getTablero(Tablero, simulation):- b_getval(simulation, Tablero).


%setters del tablero actual

setTablero(NewTablero):- getEstado(Estado), %obtiene el estado actual

                         setTablero(NewTablero, Estado). %y establece el nuevo
tablero en el estado actual

setTablero(NewTablero, tablero):- b_setval(tablero, NewTablero). %establece un
tablero nuevo, sobreescribiendo un tablero dado

setTablero(NewTablero, simulation):- b_setval(simulation, NewTablero).

tableroInicial :-

    b_setval(tablero,

            [nl,pb,nl,pb,nl,pb,nl,pb,

             pb,nl,pb,nl,pb,nl,pb,nl,

             nl,pb,nl,pb,nl,pb,nl,pb,

             va,nl,va,nl,va,nl,va,nl,

             nl,va,nl,va,nl,va,nl,va,

             pn,nl,pn,nl,pn,nl,pn,nl,

             nl,pn,nl,pn,nl,pn,nl,pn,

             pn,nl,pn,nl,pn,nl,pn,nl]), setEstado(tablero).
```

Utilidades

Durante el testeo, apunté algunos procesos que estaría bien automatizar y decidí implementarlos en forma de utilidades generales asociadas a las estructuras básicas del juego:

```
%---Utilidades---%

%%%%%%%%%%%%%%%%%

%comprueba si la posición está vacía, un ejemplo de uso es para ver si la casilla a la que va a avanzar está vacía

casillaVacia(X, Y):- getPieza(X, Y, Casilla), esVacia(Casilla).


%no hay pieza en esa posicion

noHayPieza(Posicion) :- getTablero(Tablero), nth0(Posicion, Tablero, Casilla), esVacia(Casilla).


% convierte las coordenadas X, Y en un número de dos dígitos, para así poder acceder a los elementos de la lista en la que se almacena el tablero

convertCoord(X, Y, Pos):- dentroTablero(X, Y), (Pos is ((Y-1) * 8 + (X-1))).
%calculamos la posición en la lista de la coordenada X Y, dado que el tablero posee 8 columnas, calculamos el índice en base a eso


% comprueba si las coordenadas están dentro de los límites del tablero de tamaño 8x8 (empezando en 1)

dentroTablero(X, Y) :- between(1, 8, X), between(1, 8, Y).
```

```
% función para cambiar un elemento en un array (Tablero, Índice, Nuevo Evento, Nuevo Tablero)

% intercambia el elemento del Índice del Tablero por el Nuevo Elemento y devuelve el nuevo tablero con la modificación realizada

reemplaza([_|T], 0, X, [X|T]). %estando en la posición seleccionada de la lista (el índice ha llegado a cero), empujamos el Nuevo elemento a la cabeza del Nuevo Tablero, pero en ese indice

reemplaza([H|T], I, X, [H|R]):- I > -1, NI is I-1, reemplaza(T, NI, X, R), !. % comprobamos que el índice sea mayor que -1, decrecemos el contador del índice y llamamos recursivamente a la función con la lista restante

reemplaza(L, _, _, L). %caso base, los tableros se copian


%comprueba que el jugador este en los rangos del tablero

posValida(PosOriginal, PosComida, PiezaOriginal, PiezaComida) :-

  between(0, 63, PosOriginal), %piezas en posiciones dentro del tablero

  between(0, 63, PosComida),

  PosOriginal - PosComida \= 0, % y no están en la misma posición


  %obtén las piezas en esas posiciones

  findPieza(PosOriginal, PiezaOriginal),

  findPieza(PosComida, PiezaComida),


  %comprueba si son opuestas

  piezaOpuesta(PiezaOriginal, PiezaComida).

%contador de Pieza, devuelve valor en Res

count(Pieza, Res) :- getTablero(List), countL(List, Pieza, Res, 0).
```

```
%contador de lista básico, recorre la lista y suma uno a un valor cada vez que encuentra el valor en la lista

countL( [], _, Res, Res) :- !. % final de lista

countL( [Pieza|Xs], Pieza, Res, Counter) :- !, Counter1 is Counter + 1, countL(Xs, Pieza, Res, Counter1). %si la cabeza de la lista actual coincide con la pieza, suma 1 al contador y llama recursivamente a la funcion con la cola restante

countL( [_|Xs], Pieza, Res, Counter) :- countL(Xs, Pieza, Res, Counter). %cuando el valor no coincide, se llama recursivamente a si misma con la cola restante
```

Gráficos

Para finalizar el planteamiento inicial, decidí implementar lo que sería la “interfaz” grafica, que básicamente consta de predicados que muestran cadenas por pantalla. De forma reseñable el más complejo es el que de forma recursiva recorre la lista del tablero y va mostrando el valor de cada posición, como si fuera una casilla:

```
%----Gráficos----%

%%%%%%%%%%%%%%%%%

%imprime las filas de la malla para así intentar formar el efecto visual de las casillas

printGrid :- write('|--+--+---+--+--+---+--+---+---|'), nl.


%predicado para mostrar por consola el tablero de juego actual

printTablero :- %imprime el encabezado del tablero

  write('-------------------------------------'), nl,
```

```
  write('| _ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |'), nl,
  write('|--+---+---+---+---+---+---+---+---|'), nl, printTablero(1), !. %despues, empieza a imprimir el tablero



printTablero(9) :- nl, !. %cuando termine, añade un retorno de carro


%para imprimir el tablero, solo hay que imprimir cada fila con su malla y repetirlo de forma recursiva hasta acabar el numero de filas del tablero
printTablero(Y) :- printFila(Y), printGrid, SigFila is Y + 1, printTablero(SigFila).


%para imprimir una fila, lo primero es imprimir el numero de fila, y mostrarlo en la columna 0, es decir, fuera del tablero
printFila(Y):- write('|'), printFila(Y, 0).


%imprimimos el numero de fila, lo separamos del tablero y procedvaos a imprimir los valores de las casillas hasta el 9
printFila(Y, 0):- write(Y), write(' |'), printFila(Y, 1).


%para cada casilla obtenvaos la pieza, la pasamos a un simbolo y mostramos el simbolo por pantalla. Advaas avanazamos a la siguiente columna
printFila(Y, X):- getPieza(X, Y, Pieza), simbolo(Pieza, Symbol), write(Symbol), write('|'), NextCol is X + 1, printFila(Y, NextCol).


%cnd llegamos al final de cada fila, saltamos a la siguiente
```

```
printFila(_, 9):- nl, !.


printJugadores :- write('Jugador 0: Humano'), nl,
  write('Jugador 1: IA Minimax'), nl.
```

Main

Antes de continuar, debo comentar la implementación de menú, en él se inicializa y muestra el tablero, se solicita la elección de jugadores y se emplea el predicado play que controla el flujo de los turnos hasta finalizar.

El predicado play, comprueba que se siga jugando , se realiza un turno, se muestra el movimiento a realizar, se procesa y se pasa el turno al siguiente jugador con una llamada recursiva. Cuando se detecta que no se sigue jugando, se estima el ganador y se muestra.

```
%%%%%%%%%%%%%%%%%%%%%%%%
%-----Main del Juego----%
%%%%%%%%%%%%%%%%%%%%%%%%
%inicializamos el tablero
tableroInicial :-
    b_setval(tablero,
            [nl,pb,nl,pb,nl,pb,nl,pb,
             pb,nl,pb,nl,pb,nl,pb,nl,
             nl,pb,nl,pb,nl,pb,nl,pb,
             va,nl,va,nl,va,nl,va,nl,
             nl,va,nl,va,nl,va,nl,va,
             pn,nl,pn,nl,pn,nl,pn,nl,
```

```
            nl,pn,nl,pn,nl,pn,nl,pn,
            pn,nl,pn,nl,pn,nl,pn,nl]), setEstado(tablero).


play(Color, Jugador):- %juega el jugador
    sigueJugando, write("Turno de "), write(Jugador),nl,
    turno(Jugador, Color, X, Y, NewX, NewY),
    nl, write('Movimiento: ('), write(X), write(', '), write(Y), write(') a ('),
write(NewX), write(' , '), write(NewY), write(').'), nl,
    procesarTurno(Color, X, Y, NewX, NewY),
    printTablero,
    siguienteJugador(Color, NextColor), siguienteTurno(Jugador, NextJugador),
play(NextColor, NextJugador).


play(_, _) :-
    not(sigueJugando),
    ganador(Ganador),
    write(Ganador), write(" gana! ¡Felicidades!").


main:-
  write("Bienvenido a las damas en Prolog."), nl, write("A continuación debera
elegir dos jugadores, recuerde que el primero seran las blancas y el segundo las
negras"),
  leerJugador(Jugador1, 1), setJugador(jugador1, Jugador1),
  leerJugador(Jugador2, 2), setJugador(jugador2, Jugador2),
  tableroInicial,
```

```
  printTablero,
  b_getval(jugador1, Jugador), play(blanco, Jugador).


test:- tableroInicial,
    %printTablero,
    comprobarComer(X, Y, NewX, NewY),
    write("X:"),write(X),  write("Y:"), write(Y), nl, write("NewX:"),write(NewX),
write("NewY:"), write(NewY), nl, nl.


test(Lista):-
    tableroInicial,
    %printTablero,
    findall([X, Y, NewX, NewY], comprobarJugada(blanco, X, Y, NewX, NewY),
Lista).
```

Desarrollo de la parte compleja

Lógica del Juego

Tras haber definido y probado lo que yo considero el esqueleto inicial del código necesario para poder representar y modificar la información del juego, procedí a definir la lógica del juego en términos de paso de los turnos(cuando es promocionable de dama a rey una ficha, cuando es legal un movimiento, qué rango de acción tiene) y finalización de partida(cuando se deja de jugar, cómo se decide el ganador).

Básicamente, en cada turno se comprueba quién está jugando. En caso de que sea turno de la IA, llama al algoritmo minimax para decidir qué acción realizar. En caso de que le toque a la persona, se pide por consola el movimiento que se desea realizar, en

coordenadas X,Y y se comprueba esa jugada; si la jugada no es válida, solicita que se introduzca de nuevo, mostrando un mensaje de error. Para comprobar una jugada, lo primero es ver si es legal (si ese jugador puede mover esa pieza) y después comprobar si esa jugada podría ser para comer o para desplazarse. Una vez hechas las comprobaciones, después de turno, se emplea el predicado procesar Turno, que a su vez llama al predicado comer(para intentar hacer la acción de comer una ficha) o al predicado mover(para desplazar la ficha a la casilla de destino), ambos finalizando con temp to Empty, que se encarga de convertir todas las casillas establecidas como temporales en vacías (esto se explica más adelante, en el movimiento del juego).

```
%%%%%%%%%%%%%%%%%%%%%%%%%%

%---Lógica del Juego----%

%%%%%%%%%%%%%%%%%%%%%%%%%%

%convierte los temporales en vacios

tempToEmpty:-

  getTablero(Tablero),

  nth0(Pos, Tablero, tm),

  reemplaza(Tablero, Pos, va, NewTablero),

  setTablero(NewTablero),

  tempToEmpty.


tempToEmpty.


%se obtiene el tablero (lista) y se comprueba si en la lista quedan piezas
blancas y si quedan piezas negras. Desde que un jugador se quede sin piezas,
finaliza la partida.
```

```
sigueJugando:- getTablero(Tablero), sigueJugando(Tablero, blanco),
sigueJugando(Tablero, negro).

sigueJugando(Tablero, blanco):- member(pb, Tablero), ! ; member(rb, Tablero), !.
%comprueba que queden piezas blancas (reyes o damas) en la lista que representa
el tablero

sigueJugando(Tablero, negro):- member(pn, Tablero), ! ; member(rn, Tablero), !.
%comprueba que queden piezas negras (reyes o damas) en la lista que representa el
tablero



% se decide el ganador, que es aquél que no se haya quedado sin fichas. Para ello
miramos el tablero

ganador(Ganador):- getTablero(Tablero), (ganador(Tablero, blanco, Ganador),
ganador(Tablero, negro, Ganador)).

ganador(Tablero, blanco, negro):- not(member(pb, Tablero)), not(member(rb,
Tablero)). %si no quedan piezas blancas (reyes o damas) en el tablero, ganan las
negras

ganador(Tablero, negro, blanco):- not(member(pn, Tablero)), not(member(rn,
Tablero)) . %si no quedan piezas negras (reyes o damas) en el tablero, ganan las
blancas



% promociona la pieza que este en las coordenadas X,Y a rey, mira que pieza es,
comprueba que sea promocionable, la promociona y modifica el tablero para
colocarla

promocion(X, Y):- promocionable(X, Y), procesarPromocion(X,Y).

promocion(_,_).



% comprueba si la pieza que este en X,Y puede ser promocionada

promocionable(X,Y) :-  getPieza(X, Y, Pieza), promocionable(X, Y, Pieza).
```

```
promocionable(_, Y, pb):- Y == 1.

promocionable(_, Y, pn):- Y == 8.


procesarPromocion(X, Y):- getPieza(X, Y, Pieza), promocionar(Pieza, NewPieza),
getTablero(Tablero), convertCoord(X, Y, Pos), reemplaza(Tablero, Pos, NewPieza,
NewTablero), setTablero(NewTablero).

procesarPromocion(_, _).


% comprueba si un Jugador puede jugar la pieza que se encuentra en las
coordenadas X, Y del tablero

legal(Jugador, X, Y) :- getPieza(X, Y, Pieza), legal(Jugador, Pieza).


% Primero intentamos comer, si no, Movimientomos e intentamos promocionar la
ficha

procesarTurno(_, X, Y, NewX, NewY):- comer(X, Y, NewX, NewY), promocion(NewX,
NewY), tempToEmpty.

procesarTurno(_, X, Y, NewX, NewY):- mover(X, Y, NewX, NewY), promocion(NewX,
NewY), tempToEmpty.

procesarTurno(_, _, _, _, _):- !.


%ejecuta el proceso inicial del turno en base al jugador que este jugando en este
momento

turno(humano,  Color, X, Y, NewX, NewY):- jugada(Color, X, Y, NewX, NewY).

turno(minimax,  Color, X, Y, NewX, NewY):- minimaxIA(Color, X, Y, NewX, NewY).


%realiza el movimiento que el jugador haya introducido, comprobando que sea legal
```

```
jugada(Color, X, Y, NewX, NewY):- leerJugada(X,Y,NewX,NewY),
comprobarJugada(Color, X, Y, NewX, NewY).

% si el movimiento es invalido, solicita que se introduzca de nuevo otro
movimiento

jugada(Color, X, Y, NewX, NewY):- write('Movimiento Invalido, escoge otro'), nl,
jugada(Color, X, Y, NewX, NewY).


%comprueba si una jugada es legal y factible

comprobarJugada(Jugador, X, Y, NewX, NewY):- legal(Jugador, X, Y),
(comprobarComer(X, Y, _, _, NewX, NewY); comprobarMovimiento(X, Y, NewX, NewY)).


% Lee por consola el movimiento que desea realizar el jugador

leerJugada(X,Y,NewX,NewY):- write('Introduzca la coordenada X de la ficha que
desea mover(columna)'), nl,read(X), write('Introduzca la coordenada Y de la ficha
que desea mover(fila)'), nl, read(Y), write('Introduzca la coordenada X (columna)
de la posicion hacia la que desea mover la ficha'), nl, read(NewX),
write('Introduzca la coordenada X (columna) de la posicion hacia la que desea
mover la ficha'), nl, read(NewY).
```

Movimiento del Juego

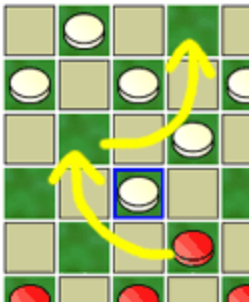
En el juego de las damas, solo hay dos posibilidades de desplazamiento, uno es un simple movimiento a una casilla adyacente en diagonal (siempre avanzando al campo contrario) y el otro es comer una pieza del jugador contrario, para ello es necesario que las fichas estén en casillas adyacentes, en la misma diagonal y que la casilla inmediatamente detrás de la ficha que va a ser comida esté vacía. Para representar toda esta lógica defini una serie de predicados para cada acción:

-Movimiento: el predicado mover, comprueba y realiza un movimiento de la pieza en X,Y a la posición NewX, NewY.

Para comprobar que el movimiento es válido, que la casillas de origen y de destino estén dentro del tablero, que la casilla de destino está vacía y por último que la pieza haga un movimiento de avance y en diagonal si es una dama, y en cualquier dirección si es un rey.

Para procesar el movimiento, simplemente coge la pieza de las coordenadas X,Y de la tabla, convierte todas las coordenadas a posiciones en la lista de la tabla y sustituye el valor en cada posición por el que corresponda (coordenada de origen vacía y coordenada de destino la pieza en cuestión). Finalmente establece el tablero modificado como el tablero efectivo.

-Comer: el predicado comer tiene dos posibilidades:



La primera, emplea comestible, que comprueba y procesa la acción de comer una ficha, seguido de una llamada recursiva a comer con nuevas coordenadas, lo que permite comprobar si se puede realizar de nuevo la acción de comer de nuevo. (Ej de situación con múltiple comida)

La segunda, solo emplea comestible para contemplar el caso en el que solo se pueda realizar la acción de comer una vez.

Para procesar la acción de comer, simplemente emplea las tres posiciones del movimiento y modifica sus valores en la tabla (Pos de origen vacía, Posición Comida vacía, Posición de destino con la pieza original).

Para comprobar la acción de comer, convierte la coordenada de origen y la de la ficha comida a posiciones, emplea un predicado con el mismo nombre pero que emplea posiciones y unifica la última variable de posición (posición de llegada) dando así los candidatos posibles (no olvidar que comer si se puede hacia atrás, así que se puede dar la situación en que existan varios candidatos a comestible que verifiquen todos los requisitos), que son comprobados inmediatamente al convertir las coordenadas de llegada a posición de llegada (deben coincidir las posición de llegada y la posición de llegada candidata).

El predicado comprobar comer que emplea posiciones, evalúa que las fichas sean de colores opuestos, no estén en la misma posición y que las posiciones estén dentro del

tablero. Seguidamente emplea un predicado que obtiene en la última variable aquellas posiciones en las quedaría después de comer fichas candidatas a ser comidas (Pos de llegada), en base al rango de acción de la pieza que vaya a comer y comprueba esa casilla de llegada esté vacía (esta comprobación no es redundante respecto a las que se realizó en los primeros predicados puesto que aquí se evalúan todas las posiciones candidatas existentes y se busca si alguna coincide con la que el usuario o la IA pretenden usar para ejecutar la acción de comer).

El predicado para comprobar piezas comestibles cercanas busca piezas en la misma diagonal, devolviendo la distancia diagonal (en filas de distancia, si están en la misma fila es 0) hasta esa pieza y el camino en casillas hasta el objetivo; se comprueba que el rango de acción de la pieza es mayor que la distancia en casillas entre la posición de origen y la de la ficha comida, se calcula el camino hasta la posición de llegada (adyacente por la diagonal a la posición de la pieza comida), se comprueba que en ese trayecto no hay piezas en medio y finalmente se calcula la posición de llegada candidata (posicion de origen - (distancia en casillas * distancia en filas)).

```
%accion de movimiento, comprueba que es legal y lo ejecuta

mover(X, Y, NewX, NewY) :- comprobarMovimiento(X, Y, NewX, NewY),
procesarMovimiento(X, Y, NewX, NewY).



%comprueba que el nuevo movimiento es legal en terminos de lugar(dentro de los
limites, casilla vacia...)

comprobarMovimiento(X, Y, NewX, NewY):- dentroTablero(X, Y), dentroTablero(NewX,
NewY), casillaVacia(NewX, NewY), getPieza(X, Y, Pieza),
comprobarMovimientoPieza(Pieza, X, Y, NewX, NewY). %pone la casilla original
vacia, mueve la ficha y guarda el mundo resultante



%procesa el movimiento, haciendolo efectivo

procesarMovimiento(X, Y, NewX, NewY):- getPieza(X, Y, Pieza), convertCoord(X, Y,
Pos), convertCoord(NewX, NewY, NewPos), getTablero(Tablero), reemplaza(Tablero,
```

```
Pos, va, NewTablero1), reemplaza(NewTablero1, NewPos, Pieza, NewTablero2),
setTablero(NewTablero2).


%comprueba los movimientos permitidos asociados a cada pieza

comprobarMovimientoPieza(pb, X, Y, NewX, NewY):-

    PermitidoX1 is X+1, PermitidoX2 is X-1, PermitidoY is Y+1, %los movimientos
de las piezas blancas son a casillas adyacentes en diagonal y avanzando hacia
abajo

    (PermitidoX1 == NewX; PermitidoX2 == NewX), PermitidoY == NewY.


comprobarMovimientoPieza(pn, X, Y, NewX, NewY):- %los movimientos de las piezas
negras son a casillas adyacentes en diagonal y avanzando hacia arriba

    PermitidoX1 is X-1, PermitidoX2 is X+1, PermitidoY is Y-1,

    (PermitidoX1 == NewX; PermitidoX2 == NewX), PermitidoY == NewY.


comprobarMovimientoPieza(rn, X, Y, NewX, NewY):- %los movimientos de las piezas
negras son a casillas adyacentes en diagonal hacia arriba o hacia abajo

    PermitidoX1 is X-1, PermitidoX2 is X+1,

    PermitidoY1 is Y-1, PermitidoY2 is Y+1,

    (PermitidoX1 == NewX; PermitidoX2 == NewX),

    (PermitidoY1 == NewY; PermitidoY2 == NewY).

comprobarMovimientoPieza(rn, _, _, _, _):- !.


comprobarMovimientoPieza(rb, X, Y, NewX, NewY):-

    PermitidoX1 is X-1, PermitidoX2 is X+1,

    PermitidoY1 is Y-1, PermitidoY2 is Y+1,
```

```
    (PermitidoX1 == NewX; PermitidoX2 == NewX),

    (PermitidoY1 == NewY; PermitidoY2 == NewY).

comprobarMovimientoPieza(rb, _, _, _, _):- !.


%accion de comer, come ficha y comprueba si en la nueva posicion de llegada puede volver a comer de forma recursiva o solo come ficha.

comer(X, Y, NewX, NewY):- comestible(X, Y, NewXTemp, NewYTemp), comer(NewXTemp, NewYTemp, NewX, NewY); comestible(X, Y, NewX, NewY).


%comprueba que la ficha sea comestible y lo procesa

comestible(X, Y, NewX, NewY) :- comprobarComer(X, Y, XComida, YComida, NewX, NewY), convertCoord(X, Y, PosOrigen), convertCoord(NewX, NewY, NewPos), convertCoord(XComida, YComida, PosComida), procesarComida(PosOrigen, PosComida, NewPos).


%procesa y reemplaza las piezas comidas por vacios temporales, mas sencillo gestionar cada turno en caso de que haya multiples comidas.

procesarComida(PosOrigen, PosComida, NewPos) :-  findPieza(PosOrigen, Pieza), getTablero(Tablero),

        reemplaza(Tablero, PosOrigen, va, NewTableroTemp), reemplaza(NewTableroTemp, PosComida, tm, NewTableroTemp2), reemplaza(NewTableroTemp2, NewPos, Pieza, NewTablero), setTablero(NewTablero).


%comprueba las coordenadas para comer

comprobarComer(X, Y, NewX, NewY):- comprobarComer(X, Y, _, _, NewX, NewY).

comprobarComer(X, Y, XComida, YComida, NewX, NewY) :- convertCoord(X, Y, PosOrigen), convertCoord(XComida, YComida, PosComida), comprobarComer(PosOrigen,
```

```
PosComida, PosLLegada), convertCoord(NewX, NewY, PosLLegada). %permite evaluar
multiples candidatos

comprobarComer(PosOrigen, PosComida, PosCandidataLlegada) :- posValida(PosOrigen,
PosComida, Pieza, _), rangoPieza(Pieza, RangoPieza),
comprobarComestiblesCercanas(PosOrigen, PosComida, RangoPieza,
PosCandidataLlegada), findPieza(PosCandidataLlegada, Casilla), esVacia(Casilla).


% Buscamos las piezas más cercanas en la misma diagonal y en el rango de accion
de la pieza

comprobarComestiblesCercanas(PosOrigen, PosEaten, RangoPieza,
PosCandidataLlegada) :-


   % estan en la misma diagonal

  piezasEnMismaDiagonal(PosOrigen, PosEaten, CaminoAlObjetivo,
DistanciaDiagonal),


  % Rango suficiente para alcanzar la pieza

  RangoPieza >= abs(CaminoAlObjetivo),


  % Get line after eatens case to use for eater after jumping over

  getCaminoNuevaPosicion(CaminoAlObjetivo, CaminoAlDestino),


  % comprobamos que no haya fichas en medio (pensado para caso en los que come el
rey desde grandes distancias)

  sinPiezasEnmedio(PosOrigen, CaminoAlObjetivo, DistanciaDiagonal,
CaminoAlDestino - CaminoAlObjetivo),
```

```
  % Posicion donde caera la ficha despues de comer
  PosCandidataLlegada is PosOrigen - DistanciaDiagonal * CaminoAlDestino.



piezasEnMismaDiagonal(PosOrigen, PosEaten, CaminoAlObjetivo, DistanciaDiagonal)
:-

  % evalua la posicion de origen con un factor de rango diagonal de +-8 (7 y 9)
  distanciaDiagonal(DistanciaDiagonal),

  % Si la posicion de la ficha comestible es mayor que la de la pieza de origen,
obtenemos un valor negativo como  camino al objetivo
  CaminoAlObjetivo is div(PosOrigen - PosEaten, DistanciaDiagonal), 0 is
mod(PosOrigen - PosEaten, DistanciaDiagonal).

distanciaDiagonal(9).
distanciaDiagonal(7).



% calculamos el camino a la posicion de llegada de la ficha despues de comer en
base al camino de la misma ficha a la ficha comestible
getCaminoNuevaPosicion(CaminoAlObjetivo, CaminoAlDestino) :-
  %en cualquier caso, ya sea con un camino al objetivo negativo o positivo, la
casilla de llegada solo esta una posicion más adelante
  CaminoAlObjetivo > 0, CaminoAlDestino is CaminoAlObjetivo + 1;
```

```
  CaminoAlObjetivo < 0, CaminoAlDestino is CaminoAlObjetivo - 1.




% Itera sobre todas las posiciones de la diagonal entre el punto de origen y el
punto de la pieza objetivo a comer, para verificar que el camino esta despejado

% y así la reina pueda comer desde largas distancias. En el caso del peon, la
primera regla se cumple por defecto asi que no necesita evaluar la diagonal

sinPiezasEnmedio(PosOrigen, CaminoAlObjetivo, DistanciaDiagonal, CaminoActual) :-

  CaminoAlObjetivo is CaminoActual;

  Posicion is (PosOrigen - CaminoActual * DistanciaDiagonal),
noHayPieza(Posicion), getCaminoNuevaPosicion(CaminoActual, NextCamino),
sinPiezasEnmedio(PosOrigen, CaminoAlObjetivo, DistanciaDiagonal, NextCamino).
```

Algoritmo MiniMax

Sin duda lo más complejo de implementar ha sido el algoritmo. Al no haber encontrado explicaciones o ejemplos detallados de la implementación de este algoritmo para este juego, empleé información del mismo pero implementado para el juego del ajedrez y luego realicé el código con las adaptaciones pertinentes al juego de las damas. Además, me sirvió mucho un video adjuntado por un compañero en el foro de elección de proyecto. De este video, adjuntado en las referencias, saqué la mayoría de ideas y recomendaciones sobre la implementación del algoritmo, como por ejemplo ideas sobre cómo simular todas las posibles jugadas para cada tablero y cada jugador en turnos alternativos, la recomendación de solo generar 3 niveles de profundidad en el árbol para mantener el programa eficiente, el cálculo del valor heurístico de los tableros almacenados en los nodos hoja, etc. Este es el predicado principal del algoritmo:

```
%algoritmo minimax
```

```
minimaxIA(Jugador, MejorX, MejorY, MejorXdest, MejorYdest):-

  initSimulationTablero, %crea una copia del tablero como simulacion en la que ejecutar los movimientos para ver el resultado

  setEstado(simulation), %establece como estado la simulacion

   getMovimientosPosibles(Jugador, MovimientosPosibles), %encuentra todos los movimientos posibles en cada momento de la simulacion

   findMinMax(Jugador, Jugador, MovimientosPosibles, [], 3, null, null, _, [MejorX, MejorY, MejorXdest, MejorYdest]), %encuentra de forma recursiva la lista que contiene el movimiento más óptimo para el momento inicial de la simulacion. Establecemos la profundidad en 3

   setEstado(tablero). %sale de la simulacion y establece el estado como el tablñero para que se puedan hacer efectivos los cambios
```

El algoritmo minmax se dedica a, para un tablero establecido como estado actual (simulacion),

```
% inicializamos el tablero simulation, con el que la IA trasteará para probar movimientos y resultados

initSimulationTablero:-

  b_getval(tablero, Tablero),

  b_setval(simulation, Tablero),

  setEstado(simulation). %establece como estado la simulacion
```

obtener la lista de todos los posibles movimientos a realizar

```
%obtiene la lista de todos los movimientos posibles para un jugador dado en el tablero actual (estado)
```

```
getMovimientosPosibles(Jugador, Lista):-
    findall([A, B, C, D], comprobarJugada(Jugador, A, B, C, D), Lista).
```

y obtener el mejor movimiento posible en base a simularlos todos.

```
simulaProximoTablero(Jugador, HistorialMovimientos):-
   % Coge el Tablero global sin movimientos simulados para procesar todos los movimientos simulados
  b_getval(tablero, InitialTablero),

  % trabajamos sobre el tablero inicial
  setTablero(InitialTablero),

   % Procesamos los movimientos de la lista de historial de movimientos para llegar hasta la situación actual
  simulaMovimientodeLista(Jugador, HistorialMovimientos).
```

Cada tablero resultante se almacena como nodo en un árbol y a su vez se repite este proceso con cada nodo del nivel en el que nos encontremos (en cada instante el nivel más profundo).

Cuando se llegue a la profundidad deseada, en este caso 3 para no generar un árbol masivo, calcula el valor heurístico de cada tablero almacenado en los nodos hoja y realiza backtracking sobre cada nivel escogiendo alternativamente el mejor y peor tablero posible de cada rama.

```
%cuando llega al nivel más profundo, calcula el valor heurístico del nodo hoja y lo devuelve (Hoja Peso, Hoja Movimiento)
```

```
findMinMax(Jugador, _, _, HistorialMovimientos, 0, _, _, HojaPeso, HojaMovimiento) :- simulaProximoTablero(Jugador, HistorialMovimientos), evaluateTablero(Jugador, HojaPeso), nth0(0, HistorialMovimientos, HojaMovimiento).


%si el se acaban los movimiento y lleva a la victoria, su peso es enorme

findMinMax(Jugador, _, [], _, _, Peso, MovimientoAProcesar, NewPeso, MovimientoAProcesar):-

  not(sigueJugando),

  ganador(Jugador),

  NewPeso is 500, !.


%en caso de que acabe la partida y no haya ganado, su peso es infimo

findMinMax(Jugador, _, [], _, _, Peso, MovimientoAProcesar, NewPeso, MovimientoAProcesar):-

  not(sigueJugando),

  NewPeso is -500, !.


%devuelve el mejor peso de esta profundidad al nivel anterior

findMinMax(_, _, [], _, _, Peso, MovimientoAProcesar, Peso, MovimientoAProcesar):- !.


% funcion recursiva que recorre el arbol

findMinMax(Jugador, Playing, [Movimiento|Tail], HistorialMovimientos, Profundidad, Peso, MovimientoAProcesar, NewMejorPeso, NewMejorMovimiento) :-
```

```
   % añade este movimiento a la lista de historial de movimientos para los proximos niveles
  append(HistorialMovimientos, [Movimiento], NewHistorialMovimientos),

  % se simula que Jugador realice el primer movimiento de la lista
  simulaProximoTablero(Jugador, NewHistorialMovimientos),

   % debemos obtener los posible siguientes movimientos, asi que simulamos que pasado el turno le toca al jugador opuesto
  siguienteJugador(Playing, SiguienteJugador),
  getMovimientosPosibles(SiguienteJugador, NewMovimientosPosibles),
  %reducimos en uno la profundidad en la que estamos buscando en el arbol para así avanzar
  NewProfundidad is Profundidad -1,

  % llamada recursiva al algoritmo para avanzar en profundidad
          findMinMax(Jugador,     SiguienteJugador,     NewMovimientosPosibles, NewHistorialMovimientos, NewProfundidad, null, null, NewPeso, NewMovimiento),

   % Compara el NewPeso encontrado en este nodo con el mejor Peso calculado respecto de los nodos anteriores
  findMejorPeso(Jugador, Playing, Peso, NewPeso, MejorPeso),

   % Selecciona el movimiento asociado al mejor peso calculado en la funcion anterior
```

```
      findMejorMovimiento(Peso,    NewPeso,    MejorPeso,    MovimientoAProcesar,
NewMovimiento, MejorMovimiento), !,



   % llamada recursiva para el siguiente nodo en este mismo nivel, por ello
empleamos el mismo valor de profundidad

     findMinMax(Jugador,   Jugador,   Tail,   HistorialMovimientos,   Profundidad,
MejorPeso, MejorMovimiento, NewMejorPeso, NewMejorMovimiento).
```

Esto es, simulando que cada nivel es un turno, que en el turno de la IA se debe escoger el tablero asociado al mejor movimiento, y que el oponente en su turno cogerá el movimiento que peor le venga a la IA. Estos dos valores son el min y el max y permiten mediante su comparación y actualización, suponiendo que el contrincante siempre jugará el mejor movimiento, encontrar el movimiento más óptimo para un instante dado, habiendo simulado los posibles siguientes 2 turnos posteriores a este movimiento.

```
%Mejor peso es la ultima variable del predicado y se unifica con el peso que
queremos establecer como el mejor

%Comparamos el nuevo peso con el peso del nivel superior,

%Si peso o NewPeso estan null, el mejor peso será aquel del que dispongamos

findMejorPeso(_, _, Peso, null, Peso):- !.

findMejorPeso(_, _, null, NewPeso, NewPeso):- !.



% En caso de que sea el mismo jugador, el MejorPeso será Peso si es mayor que
NewPeso o, NewPeso en caso contrario (es decir, escogemos el maximo peso))

findMejorPeso(Jugador, Jugador, Peso, NewPeso, Peso):- Peso > NewPeso, !.

findMejorPeso(Jugador, Jugador, Peso, NewPeso, NewPeso).
```

```
% En caso de que sea el jugador contrario, si Peso es el mayor, nos interesa que el mejor peso sea el contrario, es decir, escogemos el minimo
findMejorPeso(Jugador, _, Peso, NewPeso, NewPeso):- Peso > NewPeso, !.
findMejorPeso(Jugador, _, Peso, NewPeso, Peso).

%Si no hay movimiento a procesar(el mejor hasta el momento), el movimiento nuevo es el mejor
findMejorMovimiento(_, _, _, null, NewMovimiento, NewMovimiento).

%si peso es el MejorPeso (en este instante), el movimiento a procesar(el mejor hasta el momento) es el mejor disponible
findMejorMovimiento(Peso, _, Peso, MovimientoAProcesar, _, MovimientoAProcesar).

%si el NewPeso(calculado en este nodo) es el MejorPeso, significa que el NewMovimiento(el disponible en este nodo) es el mejor movimiento posible
findMejorMovimiento(_, NewPeso, NewPeso, _, NewMovimiento, NewMovimiento).
```

Para el cálculo heurístico, establecí que el valor de un tablero se vería dado por el número de piezas de cada jugador, siendo diferente para un mismo tablero el valor que le da cada jugador. Además, establecí el valor de los reyes como el doble del valor de los peones, como forma de ser más precisos a la hora de estimar el valor de un tablero.

```
%calcula la puntuacion de un jugador en base al numero de fichas que haya en el tablero para cada color (es decir, la puntuacion de un jugador tambien depende de las fichas de las que dispone su oponentes)
```

```
evaluateTablero(blanco, Peso) :-
  count(pb, Npb), count(rb, Nrb), %cuenta piezas blancas
  count(pn, Npn), count(rn, Nrn), %cuenta piezas negras
  Peso is (((2*Nrb) + Npb) - ((2*Nrn) + Npn)). %los reyes valen el doble, total de piezas blancas menos total de piezas negras

evaluateTablero(negro, Peso) :-
  count(pb, Npb), count(rb, Nrb), %cuenta piezas blancas
  count(pn, Npn), count(rn, Nrn), %cuenta piezas negras
  Peso is (((2*Nrn) + Npn) - ((2*Nrb) + Npb)). %los reyes valen el doble, total de piezas negras menos total de piezas blancas

%contador de Pieza, devuelve valor en Res
count(Pieza, Res) :- getTablero(List), countL(List, Pieza, Res, 0).

%contador de lista básico, recorre la lista y suma uno a un valor cada vez que encuentra el valor en la lista
countL( [], _, Res, Res) :- !. % final de lista
countL( [Pieza|Xs], Pieza, Res, Counter) :- !, Counter1 is Counter + 1, countL(Xs, Pieza, Res, Counter1). %si la cabeza de la lista actual coincide con la pieza, suma 1 al contador y llama recursivamente a la funcion con la cola restante
countL( [_|Xs], Pieza, Res, Counter) :- countL(Xs, Pieza, Res, Counter). %cuando el valor no coincide, se llama recursivamente a si misma con la cola restante
```

Este algoritmo genera un árbol de posibilidades extremadamente grande, lo cual limita severamente la profundidad, o lo que es lo mismo, cuantos turnos hacia adelante puede simular (cuantos más mejor). Para mejorar esta situación, decidí implementar una versión de este algoritmo, el que posee la poda alfa-beta; sin embargo, me quedé sin tiempo para continuar el desarrollo y decidí dejarlo como posible versión 2.0 a realizar cuando esté más libre.

Dificultades

- Implementar un predicado que calculara todas las posibles jugadas para un tablero dado
- Función recursiva que fuera escogiendo según el algoritmo minimax cada nodo
- Tener en cuenta todos los detalles y situaciones posibles a la hora de implementar este juego usando programación lógica. Tuve que realizar una lista previa de predicados y hechos posiblemente necesarios para cada momento de una partida de las damas.
- Debuggear un programa en prolog. Entre el gran número de llamadas recursivas en este tipo de algoritmos, los escuetos mensajes de error y la dificultad para establecer breakpoints útiles en prolog(terminal, no swi-Prolog web), encontrar y entender los errores que surgían se convirtió en algo bastante pesado, y eso a pesar de probar todo cada vez que se implementaba algún predicado nuevo. Prueba de ello, es el predicado test, que lo he ido modificando cada vez que saltaba un error para ir probando funcionalidades. La última en dar un error grave era la que evaluaba todas las posibles jugadas para un tablero.
- Implementar este algoritmo en prolog. Lo más cercano ha sido un código en github que empleaba un módulo ya existente como base.
- Entender qué información me faltaba en determinados momentos. Cuando desarrollas un programa así en prolog, a veces no entiendes que información puedes estar necesitando en los predicados para obtener todos los resultados esperados y no solo unos pocos.

Referencias

- https://www.youtube.com/watch?v=ipNT1QZV7Ag
- https://es.wikipedia.org/wiki/Damas
- https://wiki.ubc.ca/Chess_Game_Prolog
- https://github.com/he7850/Prolog-chess-game/blob/master/chess.pl
- http://www.fraber.de/games/chess/
- https://github.com/migafgarcia/prolog-checkers/blob/master/checkers.pl
- https://github.com/mowhebat/Checkers/blob/master/checkers.pl
- https://github.com/rstancioiu/Checkers-AI/blob/master/IA/iaminmax.pl#L10