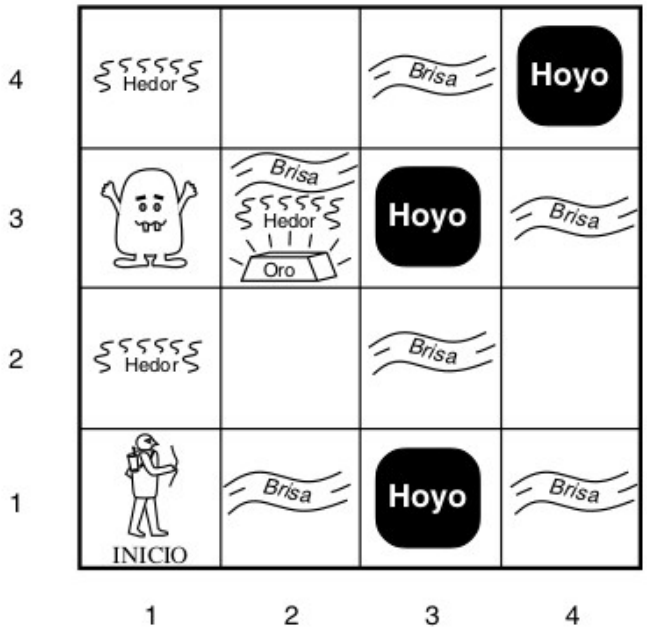
Implementación del mundo de Wumpus en prolog.
El mapa que se utilizó en la implementación del mundo de wumpus es el que se muestra de ejemplo en el libro:



en código está de la siguiente manera:

```
%El mapa para que el usuario tenga limites
mapa([_,1], sur).
mapa([_,4], norte).
mapa([1,_], oeste).
mapa([4,_], este).

%Lugares donde hay olor
olor([1,2]).
olor([1,4]).
olor([2,3]).

%Lugares donde hay brisa
brisa([2,1]).
brisa([3,2]).
brisa([4,1]).
brisa([2,3]).
brisa([3,4]).
brisa([4,3]).

%Lugar donde esta el oro
oro([2,3]).
```

Los cuales son todos los hechos del programa.

En la implementación el agente comienza mirando en la dirección del norte, cuando el agente se voltea, como salida se escribe “derecha” ó “izquierda”, y “avanza” cuando el agente se mueve de posición, de manera que si el agente está en la posición [1,1] mirando al norte, cuando avance estará en la posición [1,2] mirando igual al norte, y si se voltea a la derecha estará en la posición [1,2] mirando al este.

A continuación se muestra el resultado del programa y el tiempo en segundos que tarda el programa en ejecutarse.

El cual muestra que el programa duró 0.002245887000000002 segundos y los pasos que realizó desde la posición [1,1] mirando al norte fueron:
[avanzar,derecha,avanzar,avanzar,derecha,derecha,avanzar,avanzar,izquierda,avanzar,izquierda,avanzar,derecha,derecha,derecha,avanzar,avanzar,**recoger**,mediaVuelta,avanzar,avanzar,izquierda,izquierda,izquierda,avanzar,derecha,avanzar,derecha,avanzar,avanzar,izquierda,izquierda,avanzar,avanzar,izquierda,avanzar].
Donde recoger es cuando se recoge el oro.
Como ya sabemos prolog genera los distintos árboles de búsqueda en base a las reglas y hechos que se le dieron, por lo que al seguir ejecutando el programa me genera diferentes rutas para llegar al oro, unas más cortas que otras, como por ejemplo:

[avanzar,derecha,avanzar,derecha,avanzar,derecha,derecha,avanzar,avanzar,recoger,mediaVuelta,avanzar,avanzar,izquierda,izquierda,avanzar,izquierda,avanzar,izquierda,avanzar].

Para obtener el tiempo, se hizo uso de la función cputime de prolog, el cual regresa el tiempo en segundos.

Código comentado:
%HECHOS

%El mapa para que el usuario tenga limites
mapa([_,1], sur).
mapa([_,4], norte).
mapa([1,_], oeste).
mapa([4,_], este).

```
%Lugares donde hay olor
olor([1,2]).
olor([1,4]).
olor([2,3]).

%Lugares donde hay brisa
brisa([2,1]).
brisa([3,2]).
brisa([4,1]).
brisa([2,3]).
brisa([3,4]).
brisa([4,3]).

%Lugar donde está el oro
oro([2,3]).



%REGLAS

%Encuentra el camino que lo lleva al oro y lo regresa.

%En mover() son 5 cosas que hay que guardar, en la 1) posición es la coordenada en la que el agente está parado
% en la 2) la nueva posición a al que se movió, 3)lista que hace de las nuevas coordenadas,
%4) la lista de pasos que va haciendo el agente y el 5) lista de pasos inversos que hace el agente (para poder regresar al inicio).
muestraCamino(C) :-
        O1 is cputime,
        X = [[1,1], norte], %posición inicial
        mover(X, _, [X], L_I, L_R), %mueve al agente, L_I=lista de pasos de ida L_R=lista de pasos de
%inversos a L_I, o sea en vez de izquierda sería derecha en L_R
        regreso(L_R, [], R), %regresa al agente, (el mismo camino pero al revés)
        join(L_I, R, C), %une la lista de los pasos de ida y de regreso
        O2 is cputime, O is O2-O1, write('tiempo: '), write(O),write('segundos').


%Aquí es donde aplicamos la lógica, pues si cualquiera de las 4 coordenadas vecinas (si es que hay 4)
%de la casilla que queremos avanzar no percibe olor o brisa,
%significa que podemos avanzar ahí.
%hoyo verifica las 4 opciones, si está en el contorno del mapa también lo toma en cuenta.
hoyo([X, Y]) :-
        XO is X - 1, XE is X + 1, YS is Y - 1, YN is Y + 1,
        (brisa([XO, Y]); mapa([X,Y], oeste)),
        (brisa([XE, Y]); mapa([X,Y], este)),
        (brisa([X, YS]); mapa([X,Y], sur)),
        (brisa([X, YN]); mapa([X,Y], norte)).
```

```
%Lo mismo con wumpus
wumpus([X, Y]) :-
  XO is X - 1, XE is X + 1, YS is Y - 1, YN is Y + 1,
        (olor([XO, Y]); mapa([X,Y], oeste)),
        (olor([XE, Y]); mapa([X,Y], este)),
        (olor([X, YS]); mapa([X,Y], sur)),
        (olor([X, YN]); mapa([X,Y], norte)).

%Método para saber si es seguro avanzar a X
seguro(X) :-
        not(hoyo(X)), %si no hay wumpus o hoyo ahí, pues avanza
        not(wumpus(X)).

%El agente avanza hacie adelante si es que es seguro
avanzar([[X, Y], D], C, avanzar) :-
        (D = este, NX is X + 1, not(mapa([X, Y], D)), seguro([NX, Y]), C = [[NX, Y], D]);
        (D = norte, NY is Y + 1, not(mapa([X, Y], D)), seguro([X, NY]), C = [[X, NY], D]);
        (D = sur, NY is Y - 1, not(mapa([X, Y], D)), seguro([X, NY]), C = [[X, NY], D]);
        (D = oeste, NX is X - 1, not(mapa([X, Y], D)), seguro([NX, Y]), C = [[NX, Y], D]).

%Se voltea a la izquierda, y esto dependiendo de hacia donde está mirando el agente
izquierda([[X, Y], D], C, izquierda) :-
        (D = este, C = [[X, Y], norte]);
        (D = norte, C = [[X, Y], oeste]);
        (D = oeste, C = [[X, Y], sur]);
        (D = sur, C = [[X, Y], este]).

%Se voltea a la derecha, y esto dependiendo de hacia donde está mirando el agente
derecha([[X, Y], D], C, derecha) :-
        (D = oeste, C = [[X, Y], norte]);
        (D = norte, C = [[X, Y], este]);
        (D = este, C = [[X, Y], sur]);
        (D = sur, C = [[X, Y], oeste]).


%Los movimientos que el agente tiene
movimientos(X, M, S, O) :-
        (avanzar(X, M, S); derecha(X, M, S); izquierda(X, M, S)),
        cont(S, O). %El movimiento contrario

%¿X es miembro de la lista?
esMiembro(X, [X|_]).
esMiembro(X, [_|CL]) :-
        member(X, CL).

%Cuando ya por fin encuentra el oro
mover([X, _], [], _, [recoger, mediaVuelta], []) :-
        oro(X).
mover([X, D], [M | P], LC, [S|F], [O|R]) :-
```

```
        not(oro(X)), %si todavía no encuentra el oro
        movimientos([X, D], M, S, O), %Que movimientos puede realizar el agente, M sería la nueva coordenada
        not(esMiembro(M,LC)), %Si aún no se prueba ese movimiento, pues realizalo
        mover(M, P, [M|LC], F, R). %Ahora M es nuestra nueva coordenada de donde partimos

%Voltea la lista para tener el regreso
regreso([], X, X).
regreso([X|A], Y, R) :-
        regreso(A, [X|Y], R).

%EL típico método para juntar
join([], R, R).
join([X|F], R, [X|S]) :-
        join(F, R, S).

%ver los movimientos contrarios para el regreso
cont(avanzar, avanzar).
cont(izquierda, derecha).
cont(derecha, izquierda).
```