Programación Declarativa

Curso 2004-2005

Departamento de Electrónica, Sistemas Informáticos y Automática
Universidad de Huelva

Tema 4: Aplicaciones de Prolog.

Definición de problemas de espacio de estados

Paso previo a la búsqueda de soluciones de un problema:

- Especificación del problema

Especificar un problema como *espacio de estados* consiste en describir de manera clara cada de uno de sus componentes

Ventaja: procedimientos generales de búsqueda de soluciones *Independientes* del problema

Definición de problemas de espacio de estados

Elementos del problema

- ¿Cuál es la situación *inicial* desde la que se parte?
- ¿Cuál es el *objetivo final?*
- ¿Cómo describir las diferentes situaciones o *estados* por los que podemos pasar?
- ¿Qué acciones u *operadores* se pueden llevar a cabo en cada momento para cambiar las situaciones y cómo cambian?

Aplicaciones

- Este tipo de problemas son muy frecuentes en el campo de la inteligencia artificial. Claros ejemplos son:
  - Ajedrez
  - Puzzle
  - Grafos
  - Planificación
  - Resolución de laberintos
  - ... Etc.

Planteamiento del problema de las jarras

- Enunciado:
  - Se tienen dos jarras, una de 4 litros de capacidad y otra de 3.
  - Ninguna de ellas tiene marcas de medición.
  - Se tiene una bomba que permite llenar las jarras de agua.

- Averiguar cómo se puede lograr tener exactamente 2 litros de agua en la jarra de 4 litros de capacidad.

- Representación de estados: estado(jarra4, jarra3) con jarra4 y jarra3 el número de litros en un intervalo.
- Número de estados: 20.

Planteamiento del problema de las jarras

- Estado inicial: estado(0, 0).
- Estados finales: todos los estados de la forma estado(2, y).
- Operadores:
  - Llenar la jarra de 4 litros con la bomba.
  - Llenar la jarra de 3 litros con la bomba.
  - Vaciar la jarra de 4 litros en el suelo.
  - Vaciar la jarra de 3 litros en el suelo.
  - Llenar la jarra de 4 litros con la jarra de 3 litros.
  - Llenar la jarra de 3 litros con la jarra de 4 litros.

Planteamiento del problema de las jarras

- Aplicación de operadores a un estado (jarra4 jarra3)
  - Operador "Llenar jarra de 3"
    - Aplicabilidad: jarra3<3 (precondición)
    - Estado resultante: (jarra4 3)
  - Operador "Vaciar jarra de 4"
    - Aplicabilidad: jarra4>*0* (precondición)
    - Estado resultante: (0 jarra3)
  - Operador "Llenar jarra de 4 con jarra de 3"
    - Aplicabilidad: jarra4<4, jarra3>0, jarra4+jarra3>4 (precondición)
    - Estado resultante: (4 jarra4+jarra3-4)
- Análogamente los demás operadores

Planteamiento del problema de las jarras

- Solución:
  - Parte de un estado inicial
  - Aplica un numero finito de operadores
  - Llega al estado final

  - solucion :-

```
estadoinicial(Estado),
operaciones(Estado,Y),
estadofinal(Y).
```

Implementación Prolog

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%  Resolucion del problema del agua
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Semantica del problema:
%%   Los estados los vamos a representar como el functor
%%   estado/2
%%            estado(jarra4,jarra3).
%%   que representa la posicion de cada uno de ellos en
%%   un estado determinado.
%%
estadoinicial(estado(0,0)).
estadofinal(estado(2,_)).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% OPERACIONES DE MOVIMIENTOS
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% 1. Llenar la jarra de 4 litros de la manguera
llenar4(estado(X,Y), estado(4,Y), Li, Lo ) :-
   X < 4,
   anadiralista(Li,estado(4,Y),Lo).

% 2. Vaciar la jarra de 4 litros
vaciar4(estado(X,Y), estado(0,Y), Li, Lo ) :-
   X > 0,
   anadiralista(Li,estado(0,Y),Lo).

% 3. Llenar la jarra de 3 litros de la manguera
llenar3(estado(X,Y), estado(X,3), Li, Lo ) :-
   Y < 3,
   anadiralista(Li,estado(X,3),Lo).
```

```
% 4. Vaciar la jarra de 3 litros
vaciar3(estado(X,Y), estado(X,0), Li, Lo ) :-
        Y > 0,
        anadiralista(Li,estado(X,0),Lo).

% 5. Llenar la jarra de 4 litros con la de 3
volcar3a4(estado(X,Y), estado(Z,0), Li, Lo ) :-
        % si a la de 4 le cabe lo que hay en la de 3
        Y > 0,
        X < 4,
        4 - X >= Y,
        Z is X+Y,
        anadiralista(Li,estado(Z,0),Lo).

volcar3a4(estado(X,Y), estado(4,Z), Li, Lo ) :-
        % si a la de 4 NO le cabe lo que hay en la de 3
        Y > 0,
        X < 4,
        4 - X < Y,
        Z is Y-(4-X),
        anadiralista(Li,estado(4,Z),Lo).
```

```
% 6. Llenar la jarra de 3 litros con la de 4
volcar4a3(estado(X,Y), estado(0,Z), Li, Lo ) :-
        % si a la de 3 le cabe lo que hay en la de 4
        X > 0,
        Y < 3,
        3 - Y >= X,
        Z is X+Y,
        anadiralista(Li,estado(0,Z),Lo).

volcar4a3(estado(X,Y), estado(Z,3), Li, Lo ) :-
        % si a la de 3 NO le cabe lo que hay en la de 4
        X > 0,
        Y < 3,
        3 - Y < X,
        Z is X-(3-Y),
        anadiralista(Li,estado(Z,3),Lo).
```

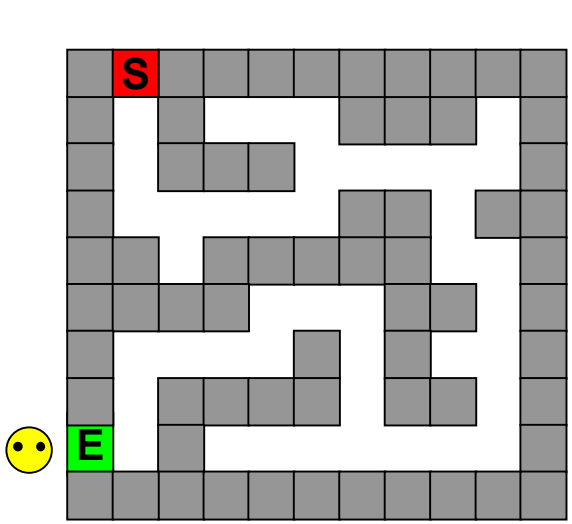
```
operacion(A,B,Li,Lo) :- llenar3(A, B, Li, Lo ).
operacion(A,B,Li,Lo) :- volcar3a4(A, B, Li, Lo ).
operacion(A,B,Li,Lo) :- llenar4(A, B, Li, Lo ).
operacion(A,B,Li,Lo) :- vaciar4(A, B, Li, Lo ).
operacion(A,B,Li,Lo) :- llenar4(A, B, Li, Lo ).
operacion(A,B,Li,Lo) :- volcar4a3(A, B, Li, Lo ).

% 5. Una operacion puede ser una sola, o un encadenamiento de ellas.
operaciones(A,B,Li,Lo) :- operacion(A,B,Li,Lo).
operaciones(A,B,Li,Lo) :- operacion(A,C,Li,L), operaciones(C,B,L,Lo).
```

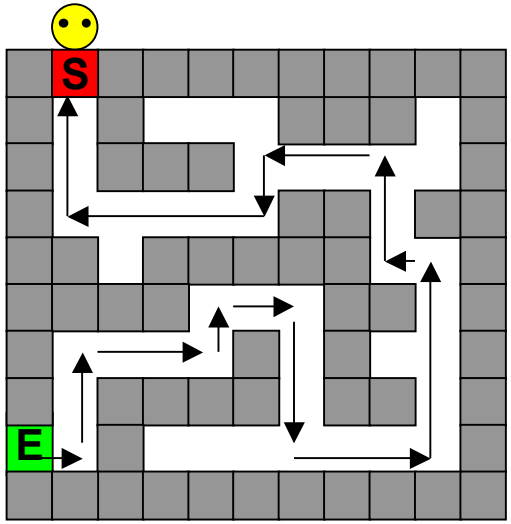
```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% BUSQUEDA DE LA SOLUCION.
%  Partiendo del estado inicial, encadenamos acciones
%  hasta llegar al estado final, y lo escribimos todo
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
solucion :-
        estadoinicial(X),
        operaciones(X,Y,[X],Lista),
        estadofinal(Y),

        write(Lista).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%
%%   Predicado auxiliar de anadiralista(A,B,C).
%%       Se añade a B a la lista A, solamente si no esta dentro
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


anadiralista(Lista,Elem,Lista) :- member(Elem,Lista),!.
anadiralista(Lista,Elem,[Elem|Lista]).
```

Laberinto

•Necesitamos que nuestro comecocos salga de un laberinto.
•El comecocos es capaz de moverse a derecha, izquierda, arriba y abajo.
•Implementar en Prolog un programa que permita a nuestro comecocos buscar la salida de un laberinto cualquiera.





Modelado del problema

Debemos elegir las representaciones adecuadas para nuestro problema:

Representaciones

¿ Cómo se representa un laberinto ?
¿ Cómo se representa al comecocos ?

Estados

¿ Cómo se define un estado en nuestro sistema ?
¿ Cuál es el estado inicial ?
¿ Cuándo un estado es final ?

Transición de estados

¿ Qué operaciones podemos aplicar para pasar de un estado a otro ?
¿ Cuáles son las restricciones del problema ?

Representaciones:

Laberinto:

El significado del laberinto que debemos codificar es el de adyacencia de posiciones y el de accesibilidad o no de las posiciones. Cualquier posible representación que recoja esta información nos es válida.

Ejemplo: una matriz de caracteres

Nota: más adelante se hacen observaciones sobre la elección de otras posibles representaciones.

Comecocos:

La información relevante del comecocos a lo largo de la resolución del problema es su posición en el laberinto.

Estados:

Los posibles estados del sistema vendrán determinados únicamente por la posición del comecocos, ya que es ésta la única información que puede variar.

El estado inicial será la entrada al laberinto.

El estado final será la salida.

Transición de estados: (transiciones posibles, operadores)

Si los estados son las posiciones, una transición de estados consiste en pasar de una posición a otra. Las posibles acciones que podemos efectuar para pasar de un estado a otro son los movimientos del comecocos: moverse arriba, moverse abajo, moverse a la derecha, moverse a la izquierda. Consideraremos los operadores 'arriba', 'abajo', 'izquierda' y 'derecha' que aplicados a un estado generan uno nuevo según las siguientes operaciones:

arriba(X1,Y1) = X1,Y2 tal que Y2 es Y1-1 (consideramos que Y crece hacia abajo)

abajo(X1,Y1) = X1,Y2 tal que Y2 es Y1+1

derecha(X1,Y1) = X2,Y1 tal que X2 es X1+1

izquierda(X1,Y1) = X2,Y1 tal que X2 es X1-1

Transición de estados: (restricciones)

Las restricciones nos marcan qué estados son válidos y qué estados no lo son. En nuestro problema, las restricciones vienen impuestas por el laberinto: no se pueden exceder las dimensiones del laberinto y nuestro comecocos sólo puede estar en posiciones que sean pasillo.

Por tanto, serán válidos los estados que verifiquen:

valido(X,Y) sii

0 <= X < columnas,
0 <= Y < filas,
laberinto(X,Y) = 'pasillo'.

Búsqueda de soluciones:

Una vez resueltas las cuestiones anteriores, la búsqueda de soluciones es una búsqueda de caminos en un grafo clásica.

La idea es:

Busca un camino:

Si el estado actual es un estado final, el camino existe y no hay que aplicar más operadores

en otro caso →

aplica un operador para pasar a otro estado
comprueba que el nuevo estado es válido
busca un camino desde el nuevo estado

Búsqueda de soluciones:

En pseudocódigo.

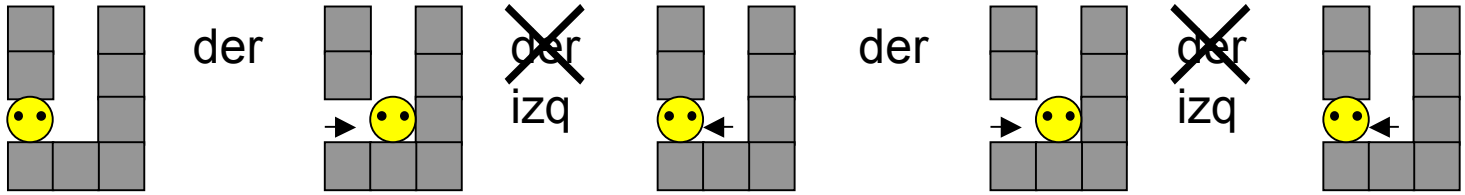
```
camino( Estado_Actual, Pasos ){

        Si final( Estado_Actual ) →
                                Pasos = {}.

        si no
                selecciona un operador OP
                Estado_Nuevo = OP( Estado_Actual ),
                valido( Estado_Nuevo ),
                inserta OP en Pasos,
                camino( Estado_Nuevo, Resto_Pasos ),
                inserta Resto_Pasos en Pasos, como siguientes a OP.
}
```

Consideraciones:

¡Atención! Con las restricciones que hemos impuesto a la transición de estados, nos exponemos a un grave problema: los ciclos.

Supongamos que aplicamos los operadores en el siguiente orden:
derecha, izquierda, arriba, abajo

Considere la siguiente situación:



¿ Cómo podemos solucionar esto ?

Consideraciones:

El problema es que estamos tomando un estado por el que ya hemos pasado como un estado nuevo y aplicándole todos los posibles operadores.

Tal como hemos construido nuestro programa, si llegamos a una posición por que ya hemos pasado, intentaremos de nuevo hacer el mismo camino que ya hicimos previamente en la recursión y entraremos en un ciclo.

La solución es la siguiente: si ya estamos buscando caminos desde una posición, no hay que intentar buscar caminos desde la misma posición de nuevo, o lo que es lo mismo: una solución de nuestro laberinto no debe pasar dos veces por la misma posición.

¿ Cómo podemos programar esto ?

Debemos observar que aquí se introduce una nueva restricción sobre los estados que son válidos. Un estado siguiente no será válido si ya ha sido visitado.

Lo que haremos es buscar un camino pero imponiendo que no pase por los nodos por los que se pasó anteriormente.

Consideraciones:

Existen diversos modos de codificar esto. Básicamente, tendremos que dar también como entrada al algoritmo de búsqueda una lista con las posiciones que ya han sido visitadas en el camino actual.

camino(Estado_Actual, Pasos, Visitados) se verificará si:

final(Estado_Actual) → Pasos = {}.
o
OP(Estado_Actual, Estado_Nuevo),
no member(Estado_Nuevo, Visitados)
valido(Estado_Nuevo),
inserta (Estado_Nuevo, Visitados, Visitados2),
camino(Estado_Nuevo, Resto_Pasos, Visitados2),
Pasos es {OP, Resto_Pasos}
}

Consideraciones:

En la implementación que se presenta, se ha optado por realizar la comprobación de si una posición ha sido ya visitada dentro del predicado que comprueba si un estado es válido:

camino(Estado_Actual, Pasos, Visitados) se verificará si:

final(Estado_Actual) → Pasos = {}.
o
OP(Estado_Actual, Estado_Nuevo),
valido(Estado_Nuevo,**Visitados**),
inserta (Estado_Nuevo, Visitados, Visitados2),
camino(Estado_Nuevo, Resto_Pasos, Visitados2),
Pasos es {OP, Resto_Pasos}
}

Implementación en Prolog:

En las siguientes transparencias se muestra el código que implementa los aspectos más importantes discutidos en las transparencias anteriores.

Representación

```
% laberintos de ejemplo
% sintaxis:
%       laberinto( Num, Lab )
%       Num --> número de laberinto
%       Lab --> matriz bidimensional de caracteres que representa al laberinto
%
%       entrada( Num, X, Y )
%       Num --> número de laberinto
%       X --> coordenada x de la entrada al laberinto Num
%       Y --> coordenada y de la entrada al laberinto Num
%
%       salida( Num, X, Y )
%       Num --> número de laberinto
%       X --> coordenada x de la salida del laberinto Num
%       Y --> coordenada y de la salida del laberinto Num
%
```

Implementación en Prolog: Representación

```
% Nota:
%           un espacio en blanco (caracter 32) representa pasillo
%           un carácter distinto del blanco representa pared

laberinto(1,[
"######## #######",
"######   #######",
"###### ##   ###",
"######    # ###",
"#   ####### ###",
"# #       ###",
"# ##############"
]).
entrada(1,7,0).
salida(1,1,6).
```

Implementación en Prolog:
Transición de estados: (transiciones posibles, operadores)

```
% Operadores de búsqueda
% sintaxis:
%           mover( Mov, X1, Y1, X2, Y2 )
%           Mov = {arr|aba|izq|der} --> nombre del movimiento
%           X1 --> coordenada X actual
%           Y1 --> coordenada Y actual
%           X2 --> coordenada X de la posición a la que se llega con el movimiento seleccionado
%           Y2 --> coordenada Y de la posición a la que se llega con el movimiento seleccionado
% Nota:
%           X1,Y1 = estado actual
%           {arr|aba|izq|der} = operadores
%           X2,Y2 = estado siguiente
mover( arr, X1, Y1, X1, Y2 ):- Y2 is Y1 - 1.
mover( izq, X1, Y1, X2, Y1 ):- X2 is X1 - 1.
mover( aba, X1, Y1, X1, Y2 ):- Y2 is Y1 + 1.
mover( der, X1, Y1, X2, Y1 ):- X2 is X1 + 1.
```

Implementación en Prolog:
Transición de estados: (restricciones)

```
% Comprobación de nodo válido
% El nodo (estado) será válido si no está fuera del laberinto, es un pasillo y no hemos pasado ya por él
% sintaxis:
%           valido( X, Y, Laberinto, Visitados )
%           X --> coordenada x
%           Y --> coordenada y
%           Laberinto --> laberinto
%           Visitados --> lista de los nodos por los que hemos pasado
valido(X,Y,Laberinto,Visitados):-filas(F), columnas(C), X>=0, Y>=0, X<C, Y<F,
   matriz( Laberinto, X, Y, 32 ), not( esta(visitado(X,Y),Visitados) ).
```

Implementación en Prolog: Búsqueda de soluciones:

```
% buscar la solución y mostrarla en pantalla
% sintaxis:
%           resuelve( Lab, Destx, Desty, Origx, Origy, Solucion )
%           Lab --> laberinto
%           Destx --> coordenada X de destino
%           Desty --> coordenada Y de destino
%           OrigX --> coordenada X de origen
%           OrigY --> coordenada Y de origen
%           Solucion --> lista de movimientos del camino
% limpiar los hechos dinámicos
resuelve(_,_,_,_,_,_):-retract( filas(_) ),fail.
resuelve(_,_,_,_,_,_):-retract( columnas(_) ),fail.
```

Implementación en Prolog: Búsqueda de soluciones:

```
resuelve( Lab, Destx, Desty, Origx, Origy, Solucion ):-
        % Calculamos el tamaño del laberinto e introducimos los hechos correspondientes
        tamanio( Lab, Filas ),
        Lab = [Fila1|_],
        tamanio( Fila1 , Cols),
        assert( filas(Filas) ),
        assert( columnas(Cols) ),
        % buscamos el camino desde el origen al destino
        % al principio sólo hemos visitado el origen
        camino( Lab, Destx, Desty, Origx, Origy, Solucion, [visitado(Origx, Origy)] ),
        % mostramos el laberinto
        muestra( Lab ),
        % mostramos la lista de movimientos
        write( Solucion ).
```

Implementación en Prolog: Búsqueda de soluciones:

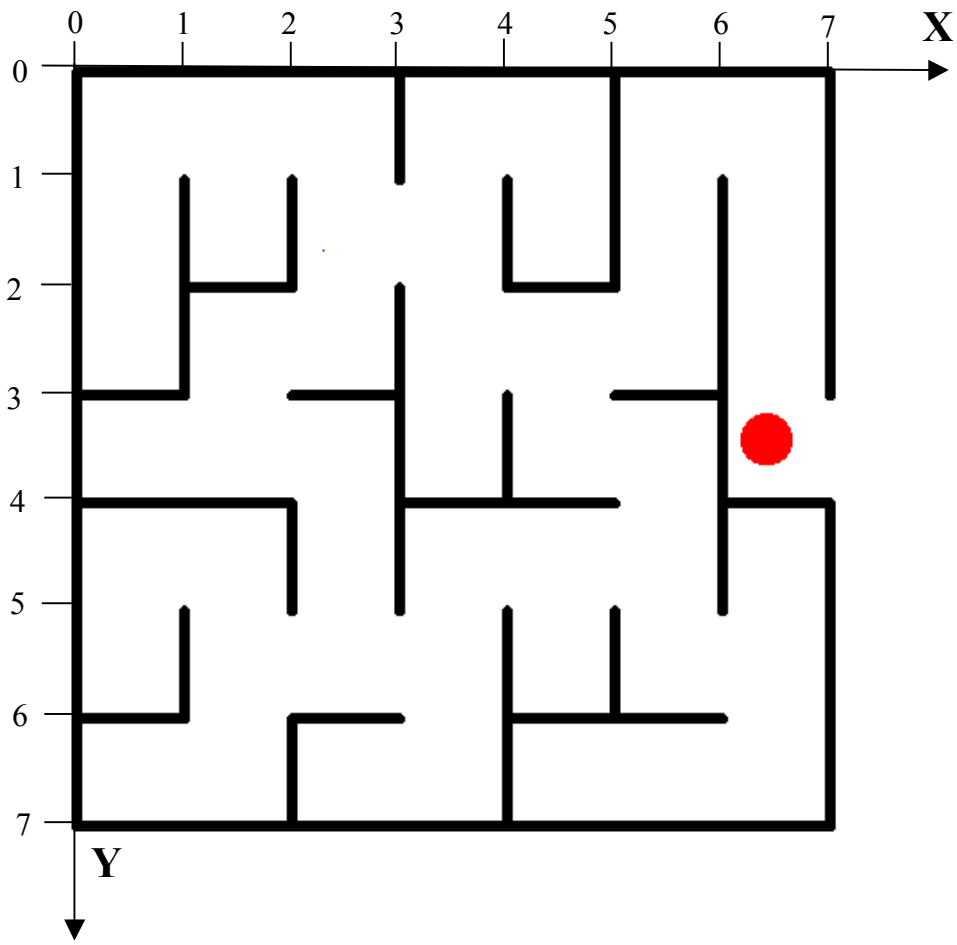
```
% Busca el camino en el laberinto, llevando cuenta de los nodos visitados para evitar bucles
%sintaxis:
%       camino( Laberinto, Objx, Objy, X, Y, Movimientos, Anteriores )
%       Laberinto --> laberinto
%       Objx --> x objetivo (destino)
%       Objy --> y objetivo (destino)
%       X --> x posición actual
%       Y --> y posición actual
%       Movimientos --> lista de movimientos necesarios para ir de la posición actual al
objetivo
%       Anteriores --> lista de posiciones por las que se ha pasado
```

Implementación en Prolog: Búsqueda de soluciones:

```
% caso base: estamos en el destino, no hay que dar ningún paso, indiferentes los visitados
camino(_, X, Y, X, Y, [], _).
% recursión
camino(Laberinto, Objx, Objy, X, Y, [Mov|Resto], Anteriores):-
        % pasar a una posición adyacente (generar un nuevo estado)
        mover(Mov,X,Y,X2,Y2),
        % comprobar que la nueva posición es válida
        valido(X2,Y2,Laberinto,Anteriores),
        % buscar un camino desde la nueva posición al destino
        camino(Laberinto, Objx, Objy, X2, Y2, Resto, [visitado(X2,Y2)|Anteriores]).
```

Ejercicios complementarios

Laberinto (Versión 2). Representación (I)




```
% Representación:
h(0,0). % fila 0
h(1,0).
h(2,0).
h(3,0).
h(4,0).
h(5,0).
h(6,0).
v(0,0).
v(3,0).
v(5,0).
v(7,0).
```

Laberinto (Versión 2). Representación (II)

- Representación de cualquier posición en el laberinto
  - posicion(X, Y)

- Representación del movimiento
  - mover(posicion(X_actual, Y_actual), posicion(X_siguiente, Y_siguiente)

Laberinto (Versión 2). Predicados auxiliares (I)

- no_visitado, visitado.
- posicion_valida.

```
visitado(posicion(X,Y), [posicion(X,Y)|_]).
visitado(posicion(X,Y), [_|Cola]):-
  visitado(posicion(X,Y), Cola).

no_visitado(posicion(X,Y), Lista) :-
  \+ (visitado(posicion(X,Y), Lista)).

posicion_valida(posicion(X,Y)):- Y >= 0, X >= 0,
  tamanio_laberinto(Filas, Columnas),
  X < Columnas, Y < Filas.
```

Laberinto (Versión 2). Predicados auxiliares (II)

- pared_arriba, pared_abajo,
  pared_derecha, pared_izquierda.

```
% pared arriba
pared_arriba(posicion(X,Y)) :- h(X,Y).

%pared abajo
pared_abajo(posicion(X,Y)) :- Y1 is Y + 1, h(X, Y1).

%pared derecha
pared_derecha(posicion(X,Y)) :- X1 is X + 1, v(X1, Y).

%pared izquierda
pared_izquierda(posicion(X,Y)) :- v(X, Y).
```

Laberinto (Versión 2). Predicados principales(I)

- mover arriba, mover abajo.

```
%mover arriba
mover(posicion(X,Y), posicion(X, Y1), Visitados):-
  Y1 is Y - 1,
  posicion_valida(posicion(X, Y1)),
  \+(pared_arriba(posicion(X,Y))),
  no_visitado(posicion(X, Y1), Visitados).
% mover abajo

mover(posicion(X,Y), posicion(X, Y1), Visitados) :-
  Y1 is Y + 1,
  posicion_valida(posicion(X, Y1)),
  \+(pared_abajo(posicion(X,Y))),
  no_visitado(posicion(X, Y1), Visitados).
```

Laberinto (Versión 2). Predicados principales(II)

- mover derecha, mover izquierda.

```
% mover derecha
mover(posicion(X,Y), posicion(X1, Y), Visitados) :-
X1 is X + 1,
posicion_valida(posicion(X1, Y)),
\+(pared_derecha(posicion(X,Y))),
no_visitado(posicion(X1, Y), Visitados).

% mover izquierda
mover(posicion(X,Y), posicion(X1, Y), Visitados) :-
X1 is X - 1,
posicion_valida(posicion(X1, Y)),
\+(pared_izquierda(posicion(X,Y))),
no_visitado(posicion(X1, Y), Visitados).
```

Laberinto (Versión 2). Predicados principales (III)

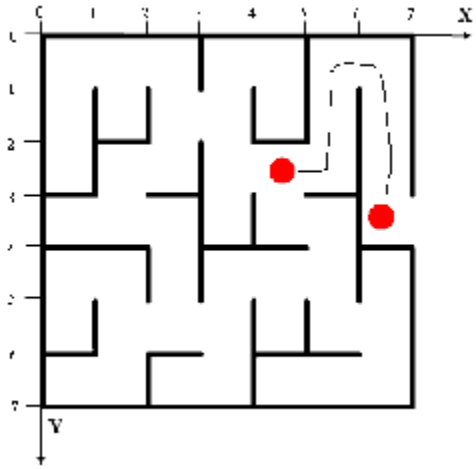
- **Camino**

camino(posicion(X_inicial, Y_inicial), posicion(X_final, Y_final), Visitados, Camino).

```
camino(posicion(X, Y), posicion(X1, Y1), Visitados,
[posicion(X,Y), posicion(X1, Y1)]):-
    mover(posicion(X, Y), posicion(X1, Y1), Visitados).

camino(posicion(X, Y), posicion(Z, T), Visitados,
[posicion(X,Y)|Camino]):-
    mover(posicion(X, Y), posicion(X1, Y1), Visitados),
    camino(posicion(X1, Y1), posicion(Z, T),
[posicion(X1, Y1)|Visitados], Camino).
```

Laberinto (Versión 2). Objetivo

```
2 ?- camino(posicion(4,2), posicion(6, 3),
[posicion(4,2)], Camino).

Camino = [posicion(4, 2), posicion(5, 2), posicion(5,
1), posicion(5, 0), posicion(6, 0), posicion(6, 1),
posicion(6, 2), posicion(6, 3)] ;

No
```

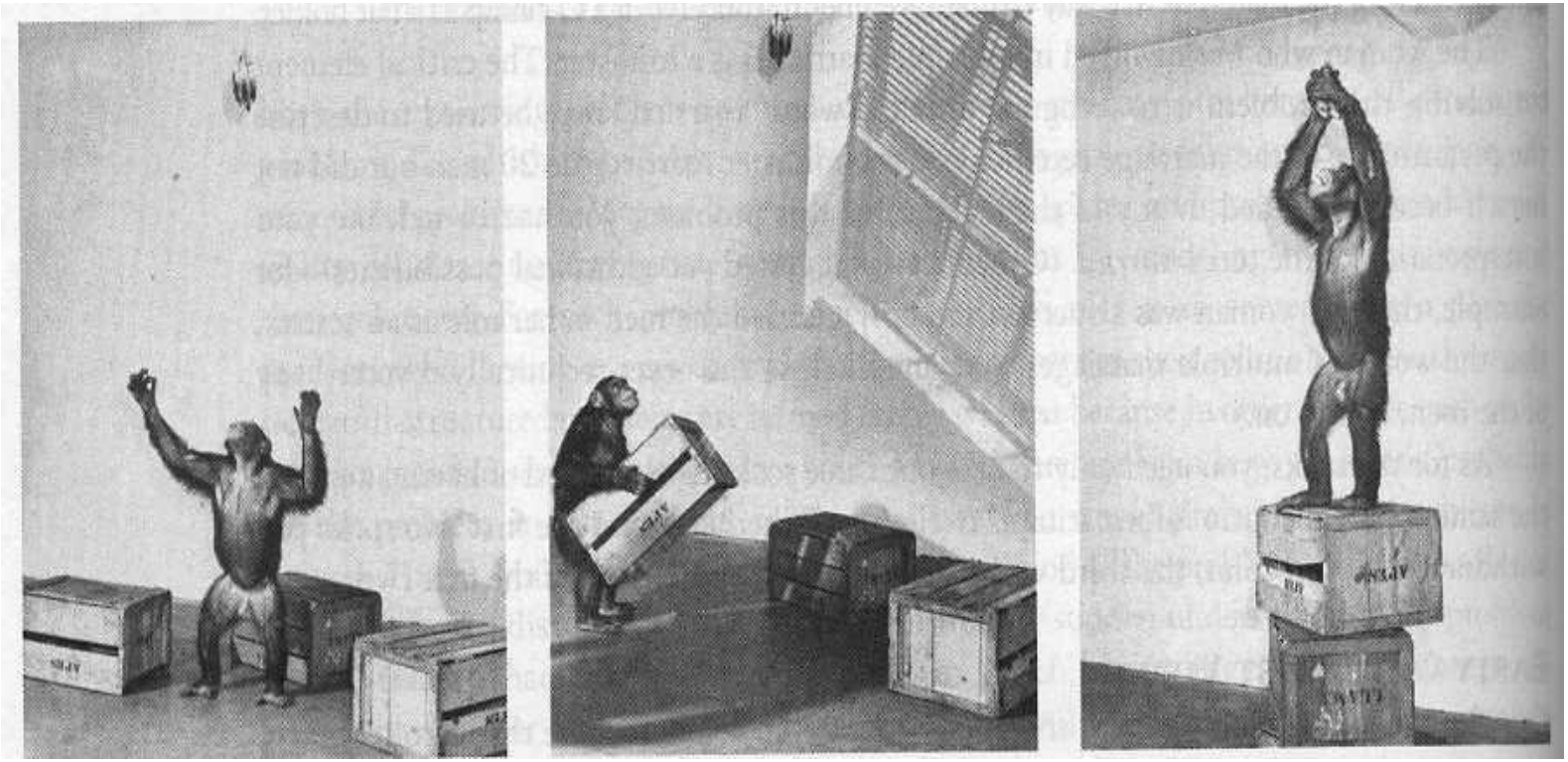


Problema del mono y la banana. Enunciado

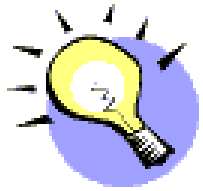
- Hay un mono en la puerta de una habitación.
- En el centro de la habitación hay una banana colgando del techo.
- El mono tiene hambre y quiere coger la banana, pero no puede saltar lo suficiente para cogerla.
- Junto a la ventana de la habitación hay una caja que el mono puede utilizar.
- El mono puede realizar la siguientes acciones: caminar sobre el suelo, subir a la caja, empujar la caja (si está junto a ella) y coger la banana si está sobre la caja justo debajo de la banana.

¿Cómo podrá el mono coger la banana?

Problema del mono y la banana. Esquema

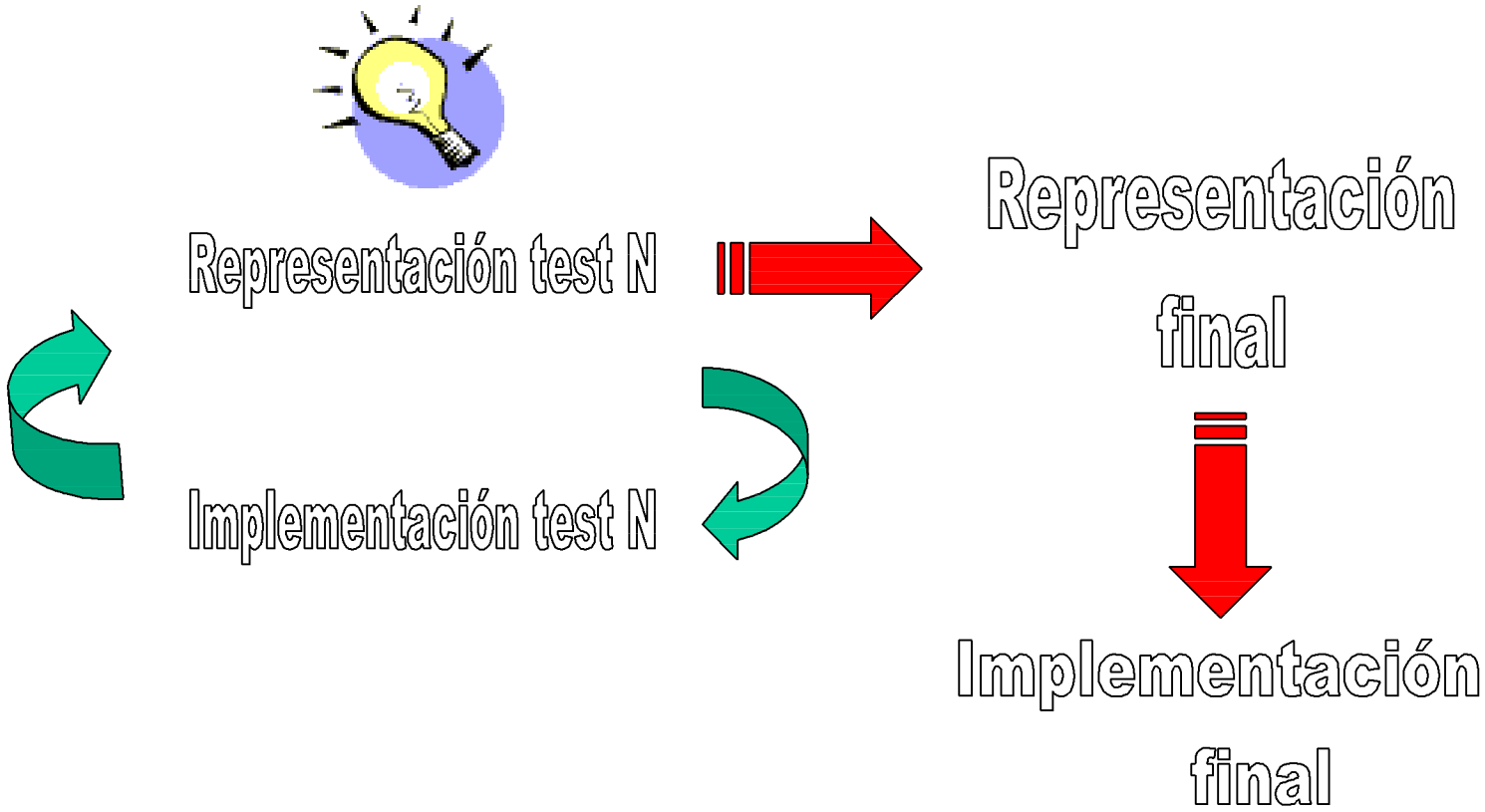


Mono y banana. Representación



- Encontrar la mejor representación es uno de los pasos más creativos en el proceso de resolución del problema.
- Elegir una buena representación es clave para resolver cualquier problema.
- Es posible que tengamos que realizar un proceso cíclico de ***representación - implementación*** hasta llegar a una buena representación del problema.

Mono y banana. Representación



Mono y banana. Representación

- Podemos definir el estado inicial de la siguiente forma:
  1) El mono está en la puerta
  2) El mono está sobre el suelo
  3) La caja está en la ventana
  4) El mono no tiene la banana

- Será necesario combinar esta información en un objeto estructurado.
- Podemos definir el funtor "estado" para representar juntas estas cuatro componentes.

Mono y banana. Representación

Representación de cualquier estado del mono

```
estado(Posicion_horizontal_mono,
  Posicion_vertical_mono,
  Posicion_caja, Banana)
```

Representación del estado inicial

```
estado(puerta, sobre_suelo,
  ventana, sin_banana)
```

Mono y banana. Representación

Posibles acciones:

1) coger banana
2) subir a la caja
3) empujar la caja
4) Caminar

- No todos los movimiento son posibles cuando se parte de un estado.
- Por ejemplo coger la banana solo se puede efectuar si el mono está sobre la caja en el centro de la habitación bajo la banana y aún no tiene la banana.

Mono y banana. Representación

Esta relación se puede representar como:

```
movimiento( Estado1, Accion, Estado2).
```

Estado1 es el estado previo a la ejecución de la acción,

Estado2 es el estado posterior a la ejecución de la acción y la Accion es alguna de los posibles acciones (coger banana, ...)

El movimiento coger_banana con sus precondiciones se puede representar del siguiente modo:

```
movimiento( estado(en_medio, sobre_caja,
  en_medio, sin_banana), coger,
            estado(en_medio, sobre_caja,
  en_medio, con_banana)
).
```

Mono y banana. Representación

De forma parecida podemos expresar el hecho de que el mono está sobre el suelo y puede caminar desde cualquier posicion Pos1 a otra posición Pos2. El mono puede caminar independientemente de la posición de la caja y tenga o no la banana.

Esto podemosexpresarlo de la siguiente forma:

```
movimiento( estado(Pos1, sobre_suelo,
  Pos_caja, Banana), caminar(Pos1, Pos2),
  estado(Pos2, sobre_suelo, Pos_caja,
  Banana)
).
```

Mono y banana. Implementación

```
movimiento( estado(Pos1, sobre_suelo,
  Pos_caja, Banana), caminar(Pos1, Pos2),
  estado(Pos2, sobre_suelo, Pos_caja,
  Banana)
).
```

Este estado representa varias cosas:

1) La acción ejecutada ha sido mover desde la Pos1 a la Pos2
2) El mono está sobre el suelo antes y después del movimiento.
3) La caja está en alguna posición Pos_caja que permanece tras el movimiento
4) El estado de la Banana, permanece después del movimiento.

Mono y banana. Implementación

¿Puede el mono desde algún estado inicial "Estado" obtener la banana?

```
puede_obtener_banana(Estado)
```

1) El predicado puede_obtener_banana será cierto si el mono ya tiene la banana. No es necesario realizar ningún movimiento en ese caso
Este hecho se corresponde con :

```
puede_obtener_banana( estado(_, _, _, con_banana)).
```

2) En cualquier otro caso uno o más movimientos serán necesarios.
El mono podrá conseguir la banana partiendo de un estado Estado1, si existe alguna acción "Accion" desde el Estado1 al Estado2 de forma que el mono pueda coger la banana en ese Estado2 (en cero o más movimientos)

```
puede_obtener_banana( Estado1) :-
  movimiento( Estado1, Accion, Estado2),
  puede_obtener_banana(Estado2).
```

Mono y banana. Implementación de movimientos

```
% Coger banana
movimiento( estado(en_medio, sobre_caja, en_medio, sin_banana),
            coger,
            estado(en_medio, sobre_caja, en_medio, con_banana)
).

% subir a la caja
movimiento( estado(Pos_horizontal_mono, sobre_suelo, Pos_caja, Banana),
            subir,
            estado(Pos_horizontal_mono, sobre_caja, Pos_caja, Banana)
).

% Empujar la caja desde la posición Pos1 a la posición Pos2
movimiento( estado( Pos1, sobre_suelo, Pos1, Banana),
            empujar(Pos1, Pos2),
            estado( Pos2, sobre_suelo, Pos2, Banana)
).

% Caminar desde la posición Pos1 a la posición Pos2
movimiento( estado(Pos1, sobre_suelo, Pos_caja, Banana),
            caminar(Pos1, Pos2),
            estado(Pos2, sobre_suelo, Pos_caja, Banana)
).
```

Mono y banana. Objetivo

```
  % estado(Posicion_horizontal_mono,
  %  Posicion_vertical_mono,
  %  Posicion_caja, Banana).

:-puede_obtener_banana(estado(puerta,
  sobre_suelo, ventana, sin_banana)).
```

```
[trace] 5 ?- puede_obtener_banana(estado(puerta, sobre_suelo, ventana, sin_banana)).
   Call: (7) puede_obtener_banana(estado(puerta, sobre_suelo, ventana, sin_banana)) ? creep
   Call: (8) movimiento(estado(puerta, sobre_suelo, ventana, sin_banana), _L202, _L185) ? creep
   Exit: (8) movimiento(estado(puerta, sobre_suelo, ventana, sin_banana), subir, estado(puerta, sobre_caja, ventana, sin_banana)) ? creep
   Call: (8) puede_obtener_banana(estado(puerta, sobre_caja, ventana, sin_banana)) ? creep
   Call: (9) movimiento(estado(puerta, sobre_caja, ventana, sin_banana), _L240, _L223) ? creep
   Fail: (9) movimiento(estado(puerta, sobre_caja, ventana, sin_banana), _L240, _L223) ? creep
   Fail: (8) puede_obtener_banana(estado(puerta, sobre_caja, ventana, sin_banana)) ? creep
   Redo: (8) movimiento(estado(puerta, sobre_suelo, ventana, sin_banana), _L202, _L185) ? creep
   Exit: (8) movimiento(estado(puerta, sobre_suelo, ventana, sin_banana), caminar(puerta, _G831), estado(_G831, sobre_suelo, ventana, sin_banana)) ?
         creep
   Call: (8) puede_obtener_banana(estado(_G831, sobre_suelo, ventana, sin_banana)) ? creep
   Call: (9) movimiento(estado(_G831, sobre_suelo, ventana, sin_banana), _L219, _L202) ? creep
   Exit: (9) movimiento(estado(_G831, sobre_suelo, ventana, sin_banana), subir, estado(_G831, sobre_caja, ventana, sin_banana)) ? creep
   Call: (9) puede_obtener_banana(estado(_G831, sobre_caja, ventana, sin_banana)) ? creep
   Call: (10) movimiento(estado(_G831, sobre_caja, ventana, sin_banana), _L257, _L240) ? creep
   Fail: (10) movimiento(estado(_G831, sobre_caja, ventana, sin_banana), _L257, _L240) ? creep
   Fail: (9) puede_obtener_banana(estado(_G831, sobre_caja, ventana, sin_banana)) ? creep
   Redo: (9) movimiento(estado(_G831, sobre_suelo, ventana, sin_banana), _L219, _L202) ? creep
   Exit: (9) movimiento(estado(ventana, sobre_suelo, ventana, sin_banana), empujar(ventana, _G839), estado(_G839, sobre_suelo, _G839, sin_banana)) ?
         creep
   Call: (9) puede_obtener_banana(estado(_G839, sobre_suelo, _G839, sin_banana)) ? creep
   Call: (10) movimiento(estado(_G839, sobre_suelo, _G839, sin_banana), _L257, _L240) ? creep
   Exit: (10) movimiento(estado(_G839, sobre_suelo, _G839, sin_banana), subir, estado(_G839, sobre_caja, _G839, sin_banana)) ? creep
   Call: (10) puede_obtener_banana(estado(_G839, sobre_caja, _G839, sin_banana)) ? creep
   Call: (11) movimiento(estado(_G839, sobre_caja, _G839, sin_banana), _L295, _L278) ? creep
   Exit: (11) movimiento(estado(en_medio, sobre_caja, en_medio, sin_banana), coger, estado(en_medio, sobre_caja, en_medio, con_banana)) ? creep
   Call: (11) puede_obtener_banana(estado(en_medio, sobre_caja, en_medio, con_banana)) ? creep
   Exit: (11) puede_obtener_banana(estado(en_medio, sobre_caja, en_medio, con_banana)) ? creep
   Exit: (10) puede_obtener_banana(estado(en_medio, sobre_caja, en_medio, sin_banana)) ? creep
   Exit: (9) puede_obtener_banana(estado(en_medio, sobre_suelo, en_medio, sin_banana)) ? creep
   Exit: (8) puede_obtener_banana(estado(ventana, sobre_suelo, ventana, sin_banana)) ? creep
   Exit: (7) puede_obtener_banana(estado(puerta, sobre_suelo, ventana, sin_banana)) ? creep
Yes
```

Planificador de vuelos

- Se pretende implementar un sistema que guarde información de los vuelos entre diferentes lugares.
- La información para cada vuelo será:
  - Origen
  - Destino
  - Número de vuelo
  - Horario de los vuelos (días de la semana, hora de salida y hora de llegada)

Planificador de vuelos. Representación

- horario(Lugar1, Lugar2, Lista_vuelos)
- Lista_vuelos::Hora_salida/Hora_llegada/Num_vuelo/Lista_dias
- Lista_dias:: [lu,ma,mi, ...]|diario

```
horario(edimburgo, londres,
     [9:40/10:50/ba4773/diario,
   13:40/14:50/ba4773/diario,
   19:40/20:50/ba4833/[lu,ma,mi,ju,vi,do]]).
```

Planificador de vuelos

- ¿Qué días de la semana hay vuelos directos desde Madrid hasta Londres?
- ¿Como puedo volar los viernes desde Madrid hasta Zurich?

¿ Como puedo visitar Milan, Madrid y Zurich, empezando en Londres el martes y volviendo a Londres el viernes con un único día en cada ciudad?

Planificador de vuelos. Predicados auxiliares

- hora_salida
- posicion_valida.
- transbordo.

```
:- op(50, xfy, :).

% hora_salida(Ruta, Hora_salida).
% La hora de salida de la Ruta es "Hora_salida"
hora_salida([_-_:_:Hora_salida|_], Hora_salida).

% Dia vuelo: Es cierto cuando Dia pertenece a la Lista_dias
%            Si utilizamos la palabra reservada "diario"
%            la lista de días será [lu, ma, mi, ju, vi, sa, do].
dia_vuelo(Dia, Lista_dias) :-
  pertenece(Dia, Lista_dias).
dia_vuelo(Dia, diario) :-
  pertenece(Dia, [lu, ma, mi, ju, vi, sa, do]).

% Al menos debemos tener 40 minutos para hacer un transbordo
transbordo(Hora1:Min1, Hora2:Min2) :-
   60* (Hora2-Hora1) + Min2-Min1 >= 40.
```

Planificador de vuelos. Predicado vuelo

Vuelo: Es cierto si existe un vuelo desde Lugar1 hasta Lugar2 el día "Dia", con un numero de vuelo “Num_vuelo" y en una Hora_salida y con una Hora_llegada.

```
vuelo(Lugar1, Lugar2, Dia, Num_vuelo, Hora_salida,
Hora_llegada) :-
  horario(Lugar1, Lugar2, Lista_vuelos),

pertenece(Hora_salida/Hora_llegada/Num_vuelo/Lista_dias,
Lista_vuelos),
  dia_vuelo(Dia, Lista_dias).
```

Planificador de vuelos. Predicado ruta

- Ruta: Predicado recursivo que obtiene la ruta entre dos lugares empezando en un día determinado. Cuando ruta se hace cierto la variable "Ruta" se instancia a la ruta entre los lugares Lugar1 y Lugar2.

```
% ruta(Lugar1, lugar2, Dia, Ruta).

% Vuelo directo
ruta(Lugar1, Lugar2, Dia, [Lugar1-Lugar2:Num_vuelo:Hora_salida], _) :-
  vuelo(Lugar1, Lugar2, Dia, Num_vuelo, Hora_salida, _).

ruta(Lugar1, Lugar2, Dia, [Lugar1-Lugar3:Num_vuelo1:Hora_salida1|Ruta],
Visitados):-
  vuelo(Lugar1, Lugar3, Dia, Num_vuelo1, Hora_salida1, Hora_llegada1),
  no_visitado(Lugar3, Visitados),
  ruta(Lugar3, Lugar2, Dia, Ruta, [Lugar3|Visitados]),
  hora_salida(Ruta, Hora_salida2),
  transbordo(Hora_llegada1, Hora_salida2).
```

Planificador de vuelos. Objetivo (I)

- ¿Qué días de la semana hay vuelos directos desde Londres hasta Madrid?

```
vuelo(londres, madrid, Dia, _, _, _).

Dia = lu ;
Dia = ma ;
Dia = mi ;
Dia = vi ;
Dia = do ;
Dia = lu ;
Dia = mi ;
Dia = ju ;
Dia = sa ;
No
```

Planificador de vuelos. Objetivo (II)

- ¿Como puedo volar los viernes desde Madrid hasta Zurich?

```
ruta(madrid, zurich, vi, Ruta, [madrid]).

Ruta = [madrid-londres:jp211:11:10,
londres-zurich:sr805:14:45] ;

No
```

Planificador de vuelos. Objetivo (III)

¿ Como puedo visitar Milan, Madrid y Zurich, empezando en Londres el martes y volviendo a Londres el viernes con un único día en cada ciudad?

```
permutation([milan, madrid, zurich], [Ciudad1, Ciudad2,
Ciudad3]),
vuelo(londres, Ciudad1, ma, Vuelo1, Sal1, Lleg1),
vuelo(Ciudad1, Ciudad2, mi, Vuelo2, Sal2, Lleg2),
vuelo(Ciudad2, Ciudad3, ju, Vuelo3, Sal3, Lleg3),
vuelo(Ciudad3, londres, vi, Vuelo4, Sal4, Lleg4).
```

Planificador de vuelos. Objetivo (III)

¿ Como puedo visitar Milan, Madrid y Zurich, empezando en Londres el martes y volviendo a Londres el viernes con un único día en cada ciudad?

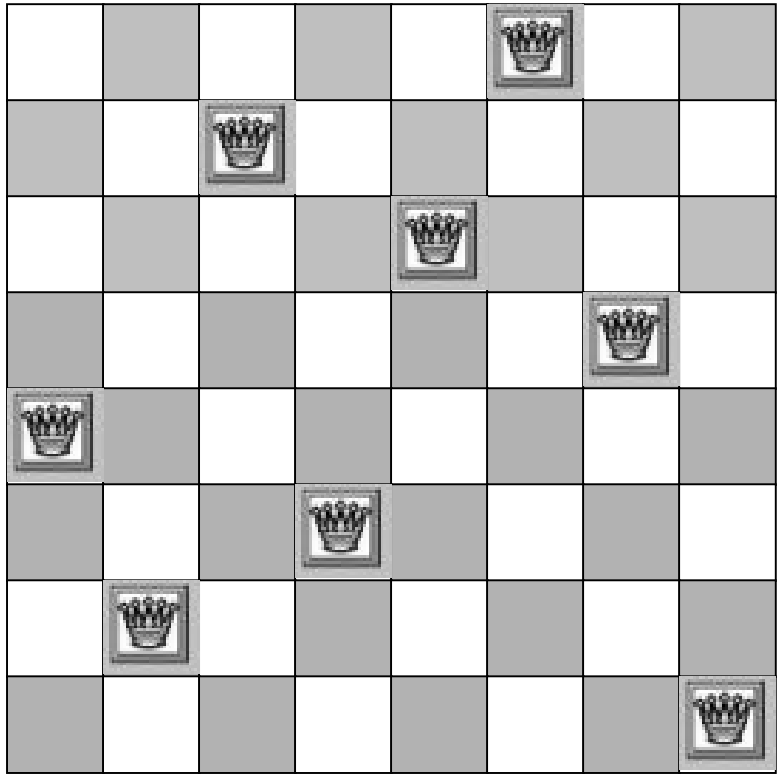
```
Ciudad1 = milan
Ciudad2 = zurich
Ciudad3 = madrid
Vuelo1 = ba510
Sal1 = 8:30
Lleg1 = 11:20
Vuelo2 = sr621
Sal2 = 9:25
Lleg2 = 10:15
Vuelo3 = jp323
Sal3 = 13:30
Lleg3 = 14:40
Vuelo4 = jp211
Sal4 = 11:10
Lleg4 = 12:20
```

Planificador de vuelos.

```
horario( edimburgo, londres,
           [  9:40 / 10:50 / ba4733 / diario,
             13:40 / 14:50 / ba4773 / diario,
             19:40 / 20:50 / ba4833 / [lu,ma,mi,ju,vi,su] ] ).
horario( londres, edimburgo,
           [  9:40 / 10:50 / ba4732 / diario,
             11:40 / 12:50 / ba4752 / diario,
             18:40 / 19:50 / ba4822 / [lu,ma,mi,ju,vi] ] ).
horario( londres, madrid,
           [ 13:20 / 16:20 / jp212 / [lu,ma,mi,vi,do],
             16:30 / 19:30 / ba473 / [lu,mi,ju,sa] ] ).
horario( londres, zurich,
           [  9:10 / 11:45 / ba614 / diario,
             14:45 / 17:20 / sr805 / diario ] ).
horario( londres, milan,
           [  8:30 / 11:20 / ba510 / diario,
             11:00 / 13:50 / az459 / diario ] ).
horario( madrid, zurich,
           [ 11:30 / 12:40 / jp322 / [ma,ju] ] ).
horario( madrid, londres,
           [ 11:10 / 12:20 / jp211 / [lu,ma,mi,vi,do],
             20:30 / 21:30 / ba472 / [lu,mi,ju,sa] ] ).
horario( milan, londres,
           [  9:10 / 10:00 / az458 / diario,
             12:20 / 13:10 / ba511 / diario ] ).
horario( milan, zurich,
           [  9:25 / 10:15 / sr621 / diario,
             12:45 / 13:35 / sr623 / diario ] ).
horario( zurich, madrid,
           [ 13:30 / 14:40 / jp323 / [ma,ju] ] ).
horario( zurich, londres,
           [ 9:00 / 9:40 / ba613 / [lu,ma,mi,ju,vi,sa],
            16:10 / 16:55 / sr806 / [lu,ma,mi,ju,vi,do] ] ).
horario( zurich, milan,
           [ 7:55 / 8:45 / sr620 / diario ] ).
```

Enunciado del problema de las 8 reinas

Colocar 8 reinas en un tablero rectangular de dimensiones 8x8, de forma que no se encuentren más de una en la misma línea horizontal, vertical o diagonal.



Representación del problema

Opción inicial:

- La primera opción es representar una lista con cada una de las posiciones de las reinas.

[[x1,y1], [x2,y2], [x3,y3], ...]

Esta representación tiene la facilidad de compresión, ya que cada uno los elementos de la lista son una posición en un espacio de coordenadas.

Representación del problema

Acercándonos un poco mas al problema:

Es imposible que haya dos reinas en la misma fila o columna

Por lo que, podemos representar el estado como una lista donde hacemos referencia únicamente a la columna(fila) en la que se encuentra.

estado = [6,3,5,7,1,4,2,8].

Solución del problema

Una forma de solucionar el problema puede ser:

- Ir colocando una a una las reinas en el tablero, de forma que las posiciones vayan siendo seguras

La forma de implementar esto puede ser una recursión hasta vaciar la lista de reinas e ir llenando la lista en las posiciones seguras.

```
solucion1([]).
solucion1([Y | Otras]) :-
        solucion1(Otras),
        member(Y,[1,2,3,4,5,6,7,8]),
        no_ataque(Y,Otras).
```