Convenios de representación en Prolog: paradigma del espacio de estados

- Fijar la **representación de los estados** y escribirlos en Prolog mediante una estructura construida con un functor *estado/n*, cuya aridad dependerá del problema:

```
estado(...)
```

  No es un predicado, sino términos que usaremos como argumentos de otros predicados.

- Explicitar cuál es el **estado inicial** mediante un predicado *inicial/1*:

```
inicial(Estado).
```

- Explicitar cuáles son los **estados objetivos** mediante un predicado *objetivo/1*:

```
objetivo(Estado).
```

- Explicitar cuáles son los **operadores** mediante un predicado *movimiento/4*:

```
movimiento(Estado, EstadoSiguiente, CosteOper, NombreOper) :-
                    Especificación.
```

- Usaremos tantos **predicados auxiliares** como se considere adecuado para obtener el código más claro y legible posible.

- Si en nuestro problema hay **situaciones de peligro** correspondientes a estados que debemos evitar generar, utilizaremos un predicado *peligro/1*:

```
peligro(Estado) :- Condición.
```

  para explicitar qué estados están en situación de peligro.

  Esta comprobación a veces aparece en los algoritmos de búsqueda y otras en la especificación de los operadores. Nosotros la especificaremos aparte, pero la añadiremos a los operadores de cara a evitar generar estados de peligro en el proceso de búsqueda. Así, la búsqueda no llegará a generar el estado de peligro para desecharlo después (cosa que ocurriría si incluyésemos la comprobación en el algoritmo de búsqueda).

- Si tenemos previsto usar algún algoritmo de búsqueda informada que trabaje con información heurística, explicitaremos cuál es la **función heurística** que vamos a emplear mediante un predicado *heurística/2*:

```
heuristica(Estado, MedidaHeurística) :- ...
```

  Si el número de estados es manejable y la heurística no puede describirse de otra manera más concisa (con reglas que afecten a más de un estado), será una colección de hechos.

  Los algoritmos de búsqueda informada utilizarán este predicado.

- Para **resolver un problema** necesitaremos utilizar alguna estrategia de búsqueda. Por ello, hemos de determinar cómo ha de interactuar un problema con una estrategia de búsqueda. Cargaremos el fichero que nos interese, dependiendo de la estrategia elegida con el predicado Prolog *ensure_loaded/1*. Por ejemplo:

```
:- ensure_loaded('profundidad').
```

  Los algoritmos de búsqueda tendrán implementado el predicado *resuelve/1*, que usaremos para resolver cada problema a través del predicado *test/0*:

```
test :- inicial(EstadoInicial), resuelve(EstadoInicial).
```

  La búsqueda con profundidad limitada debe usar el predicado *resuelve/2* para pasarle el límite que queramos fijar en el intento de búsqueda de una solución a nuestro problema:

```
test(Límite) :- inicial(EstadoInicial),
                    resuelve(EstadoInicial, Límite).
```

- En general, los algoritmos de búsqueda tendrán implementado el predicado *resuelve/1* que necesitamos para resolver los problemas:

```
resuelve(EstadoInicial) :- ...
```

  desde el cual se procederá a llamar al predicado encargado de la búsqueda de un camino a la solución, el cual dependerá de la estrategia de búsqueda.

Implementación en Prolog de estrategias de búsqueda en el espacio de estados

- Las estrategias que realizan una búsqueda de tipo primero en profundidad pueden implementarse de forma especialmente sencilla en Prolog, aprovechándose de que la búsqueda que realiza Prolog es justamente de ese tipo y del control que lleva Prolog para el proceso de vuelta atrás. Por ejemplo, la **búsqueda primero en profundidad tirando de Prolog** podría implementarse como sigue:

```
% Descomentar para uso en SISCtus Prolog (en SWI-Prolog, no)
% :- use_module(library(lists)).   % para member

% Comienzo de la búsqueda
resuelve(Inicial) :-
        write('Búsqueda PRIMERO EN PROFUNDIDAD (tirando de Prolog ; '),
        write('control de ciclos en camino actual)'), nl,
        camino(Inicial, [Inicial], Camino, Operadores),
        write('SOLUCIÓN ENCONTRADA A PROFUNDIDAD: '),
        imprime_camino(Camino, -1),
        write('OPERADORES UTILIZADOS: '), nl, write(Operadores).

% Búsqueda de solución
camino(Estado, Camino, Camino, []) :- objetivo(Estado), !.

camino(Estado, CaminoActual, CaminoFinal, [Operador| Operadores]) :-
        movimiento(Estado, Siguiente_Estado, _Coste, Operador),
        % procesa nodo: se genera un hijo
        \+(member(Siguiente_Estado, CaminoActual)),
        % si hijo no visitado en el camino actual
        camino(Siguiente_Estado, [Siguiente_Estado|CaminoActual],
                                        CaminoFinal, Operadores).
        % añade hijo al camino actual y prosigue la búsqueda

% Imprime una lista en orden inverso
imprime_camino([], Profundidad) :- write(Profundidad), nl.
imprime_camino([E|Resto], Profundidad) :-
        P2 is Profundidad + 1, imprime_camino(Resto, P2),
        write(E), nl.
```

El **orden de generación de hijos** se corresponde con la aplicación de los operadores en el **orden textual** en el que aparecen en la representación del problema.

Los estados que se almacenan **sólo** son los correspondientes al **camino actual** (el que se está siguiendo en la búsqueda de la solución). Con ello, **se controla sólo en parte la repetición de estados**, pero se posibilita el revisitar estados de caminos abandonados anteriormente, y no se lleva control de los pendientes de expandir *(abiertos)*.

Se tira de Prolog para el ***backtracking***: cada vez que se procesa un nodo se genera sólo el primer hijo, que es por el que prosigue la búsqueda. El proceso de vuelta atrás lo provoca y controla Prolog que vuelve al último punto de elección (nuevo hijo del nodo más cercano con hijos no generados pendientes de generar y visitar).

Si *camino/4* no tiene éxito, se produce un fallo lo cual supone que Prolog hace *backtracking* de forma automática al último punto de elección pendiente. Si no hubiese ninguno pendiente, la búsqueda de camino fallaría.

- No obstante, la implementación de otras estrategias de búsqueda o la necesidad de hacer un control de repetición de estados más completo hace aconsejable considerar otras alternativas de implementación más generales en las que sea nuestro programa el que controle el orden de exploración de nodos, así como el proceso de vuelta atrás. Esto se realiza almacenando los nodos pendientes de visitar en una estructura. Por ejemplo, la **búsqueda primero en profundidad** (sin tirar de Prolog) podría implementarse como sigue:

```
% Descomentar para uso en SISCtus Prolog (en SWI-Prolog, no)
% :- use_module(library(lists)).       % permite usar member, length y append

% Comienzo de la búsqueda
resuelve(Inicial) :-
        write('BÚSQUEDA PRIMERO EN PROFUNDIDAD '),
        write('(control de ciclos: camino actual)'), nl,
        camino([(Inicial, nil, _)], []).

% Búsqueda de camino
camino(Abiertos, _) :-
        Abiertos = [], !,    % no quedan nodos que visitar
        write('NO SE ENCONTRÓ NINGUNA SOLUCIÓN'), nl.

camino(Abiertos, Cerrados) :-
        Abiertos = [(Estado, Padre, Operador)|_],  % extrae nodo
        objetivo(Estado), !,                       % si es estado objetivo
        write('SOLUCIÓN ENCONTRADA'),
        imprimeSolucion((Estado, Padre, Operador), Cerrados, 0), nl,
        write('Nodos expandidos: '), length(Cerrados, L), write(L), nl.

camino(Abiertos, Cerrados) :-
        Abiertos = [(Estado, Padre, Operador)|Resto_Abierta], % extrae nodo
        obten_hijos(Estado, Padre, Cerrados, Hijos),           % expande nodo
        append(Hijos, Resto_Abierta, N_Abiertos),
        % añade sus hijos a los pendientes de visitar (candidatos)
        pon_en_cjto((Estado, Padre, Operador), Cerrados, N_Cerrados),
        % marca estado del nodo como visitado
        camino(N_Abiertos, N_Cerrados).
        % prosigue la iteración
```

C*amino/2* termina su ejecución cuando se llega a un estado objetivo y se ha encontrado un camino, o bien cuando no quedan nodos pendientes de visitar, en cuyo caso la búsqueda de camino habrá fallado.

No se tira de Prolog para el ***backtracking***, sino que cada vez que se procesa un nodo se generan todos los hijos, los cuales se almacenan en los pendientes de visitar en forma de ternas *(nodo_hijo, nodo_padre, operador)*. Para ello, se utiliza una pila (abiertos) que es la estructura que marca el orden de exploración (LIFO). Además, se almacenan también los nodos ya expandidos en un conjunto (cerrados).

El **orden de generación de hijos** se corresponde con la aplicación de los operadores en el **orden textual** en el que aparecen en la representación del problema.

```
% Generación de todos los hijos de un nodo: expandir nodo
obten_hijos(Estado, Padre, Cerrados, Hijos) :-
        findall(Hijo, movimientos(Estado, Padre, Cerrados, Hijo), Hijos).

movimientos(Estado, Padre, Cerrados, (Siguiente, Estado, Operador)) :-
        movimiento(Estado, Siguiente, _, Operador),
        % Control de ciclos: no en camino actual
        \+(enCaminoActual(Siguiente, Padre, Cerrados)).

% Controla si un estado ha aparecido previamente en el camino actual.
% El camino actual se puede reconstruir a partir de Cerrados
enCaminoActual(Estado, Estado, _).
enCaminoActual(Estado, Antecesor, Cerrados) :-
                        member((Antecesor, NAnt, _), Cerrados),
                        enCaminoActual(Estado, NAnt, Cerrados).


% Predicados auxiliares
imprimeSolucion((Estado, nil, _), _, Profundidad) :-
        !, write(' A PROFUNDIDAD: '), write(Profundidad), nl,
        write('Estado: '), write(Estado).

imprimeSolucion((Estado, Padre, Operador), Cerrados, Profundidad) :-
        member((Padre, Abuelo, Op2), Cerrados),
        P2 is Profundidad + 1,
        imprimeSolucion((Padre, Abuelo, Op2), Cerrados, P2),
        tab(3), write('Operador: '), write(Operador), nl,
        write('Estado: '), write(Estado).

% Añadir un elmento a un conjunto (sólo si no está ya):
pon_en_cjto(X, S, S) :- member(X, S), !.
pon_en_cjto(X, S, [X|S]).
```