Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Inteligencia Artificial:

Proyecto Individual Prolog.

Solución en Prolog del problema del lobo, la oveja y la col.

José Javier Díaz González
(alu0101128894@ull.edu.es)

Índice

1. Introducción

Este documento describe las funcionalidades y el razonamiento detrás del código, así como las posibles dificultades encontradas a lo largo de su desarrollo.

La práctica fue desarrollada por José Javier Díaz González (alu0101128894@ull.edu.es)

2. Objetivos de la práctica

El objetivo de este proyecto es la realización de forma individual, en este caso, mi proyecto es un sistema basado en el conocimiento en Prolog para dar solución al problema del lobo, la oveja y la col.

El enunciado del problema de esta práctica es el siguiente:

(Sacado de google)

"Un pastor debe transportar un lobo, una oveja y una col a través de un río, usando una barca en la que sólo cabe él junto a uno de ellos. El lobo y la oveja no pueden quedarse solos en la misma orilla sin el pastor, ni la oveja puede quedarse sola con la col. El pastor puede hacer tantos viajes a través del río como sea necesario."

3. Código

```
extraer(X, [X|L], L).
extraer(X, [Y|L], [Y|R]):-not(X=Y), extraer(X,L,R).

pertenece(X, [X|_]).
pertenece(X, [_|Y]) :- pertenece(X,Y).
perteneceLista([], _).
perteneceLista([H|T], L):- pertenece(H,L), perteneceLista(T,L).

insertar(X,L,[X|L]).

append([],L,L).
append([H|T],L2,[H|L3]):-append(T,L2,L3).

viajar(OE1,OE2,A,OS1,OS2):-
    extraer(A, OE1, OS1),
    insertar(A, OE2, OS2).
/* Condiciones de Movimientos */
/* IZQUIERDA A DERECHA */

mover(OE1,OE2,OS1,OS2):- pertenece(pastor,OE1), pertenece(oveja,OE1),
    viajar(OE1,OE2,oveja,Q,P), viajar(Q,P,pastor,OS1,OS2).

mover(OE1,OE2,OS1,OS2):-pertenece(pastor,OE1),pertenece(col,OE1),
    viajar(OE1,OE2,col,Q,P), viajar(Q,P,pastor,OS1,OS2),
    not(perteneceLista([oveja,lobo],OE1)).

mover(OE1,OE2,OS1,OS2):-pertenece(pastor,OE1), pertenece(lobo,OE1),
    viajar(OE1,OE2,lobo,Q,P), viajar(Q,P,pastor,OS1,OS2),
    not(perteneceLista([oveja,col],OE1)).

mover(OE1,OE2,OS1,OS2):-viajar(OE1,OE2,pastor,OS1,OS2),
    not(perteneceLista([oveja,col],OE1)) | not(perteneceLista([oveja,lobo],OE1)).
/* DERECHA A IZQUIERDA */

mover(OE1,OE2,OS1,OS2):- pertenece(pastor,OE2), pertenece(oveja,OE2),
    viajar(OE2,OE1,oveja,Q,P), viajar(Q,P,pastor,OS2,OS1).

mover(OE1,OE2,OS1,OS2):- pertenece(pastor,OE2), pertenece(col,OE2),
    viajar(OE2,OE1,col,Q,P), viajar(Q,P,pastor,OS2,OS1),
    not(perteneceLista([oveja,lobo],OE2)).

mover(OE1,OE2,OS1,OS2):- pertenece(pastor,OE2), pertenece(lobo,OE1),
    viajar(OE2,OE1,lobo,Q,P), viajar(Q,P,pastor,OS2,OS1),
    not(perteneceLista([oveja,col],OE2)).

mover(OE1,OE2,OS1,OS2):-viajar(OE2,OE1,pastor,OS2,OS1),
    not(perteneceLista([oveja,col],OE2)) | not(perteneceLista([oveja,lobo],OE2)).
/* Solución */

sol([],_,ACC,ACC).
sol(OE1, OE2, ACC, RESULT):- mover(OE1,OE2,OS1,OS2), not(pertenece([OS1,OS2],ACC)),
    append(ACC,[[OS1,OS2]],ACC2),sol(OS1,OS2,ACC2,RESULT).
```

4. Desarrollo del código

En este apartado se comentará las funcionalidades del código desde el punto de vista de una breve introducción teórica sobre las funcionalidades y el porqué de su implementación.

4.1. Funcionalidad del código

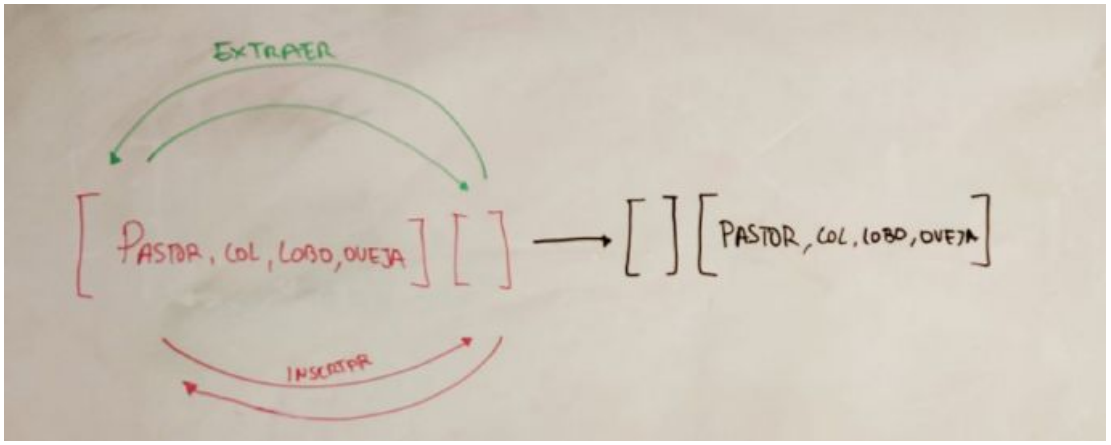

Figura 1

Al comienzo de la práctica, antes de diseñar e implementar la misma, extraje de forma superficial el concepto del problema. Desde un comienzo, el enunciado del problema nos dice que vamos a transportar diversos elementos de una orilla a otra donde solo hay espacio para transportar un elemento más el pastor con una serie de restricciones. Esto quiere decir que la función principal del problema es transportar elementos de una orilla a otra. Las orillas las vamos a representar con listas por lo tanto, cuando transportamos elementos de una orilla a otra vamos a extraer e insertar de una lista a otra. La finalidad del problema es transportar todos los elementos hasta la otra orilla por lo que continuaremos estos movimientos hasta que transportemos todos los elementos de una orilla a la otra. En la **figura 1**, se puede ver de forma visual un esquema de lo que he comentado.

Extraer e insertar elementos de una lista(orilla) a otra. Por consiguiente necesitamos dos métodos que nos realicen esta función: insertar y extraer.

En prolog cuando queremos realizar alguna función que retorne algo, le tenemos que pasar como parámetro una variable donde se almacene, además de los parámetros de la propia función.

```
insertar(X,L,[X|L]).
```

A la función insertar le pasamos el elemento que queremos insertar (**X**) , la lista donde la vamos a insertar (**L**) y nos devuelve un resultado(**[X|L]**) que es una lista que contiene al elemento(**X**) en la cabeza y a la lista donde queremos insertar el elemento(**L**). Es decir, nos inserta el elemento por la cabeza.

```
extraer(X, [X|L], L).
extraer(X, [Y|L], [Y|R]):-not(X=Y), extraer(X,L,R).
```

A la función extraer le pasamos el elemento que queremos extraer(**X**), la lista donde lo vamos a extraer([**X|L**]) y nos devuelve la lista resultado(**L**) de la cual fue extraído el elemento.

En la primera línea encontramos la solución base de extraer. Si el elemento se encuentra en la cabeza basta con retornar la cola. Pero ¿Qué pasa cuando el elemento a extraer no está en la cabeza? Pues vamos desplazando la cabeza hasta recorrer la lista completa. Si el elemento se encuentra en la lista lo encontraremos a medida que vamos desplazando los elementos y convirtiéndolos en la cabeza. Una vez encontrado el elemento retornamos todo lo demás.

En este punto soy capaz de extraer e insertar en una lista, pero la función que hacemos en concreto en el problema es transportar que consiste en extraer de una lista e insertar en otra. Por lo tanto, necesitamos un método o función que nos permita realizar estos dos pasos de una sola vez.

```
viajar(OE1,OE2,A,OS1,OS2):-
    extraer(A, OE1, OS1),
    insertar(A, OE2, OS2).
```

Nuestra función que se encarga de transportar los elementos es viajar. Viajar tiene como parámetros la orilla izquierda(**OE1**), la orilla derecha(**OE2**), el elemento a mover(**A**) y como hemos comentado anteriormente dos parámetros donde se almacena las listas (orillas) modificadas una vez extraído (**OS1**) e insertado el elemento (**OS2**). Ahora podemos poner en práctica transportar elementos de una orilla a otra. Cuando le pasemos estos paŕametros extraerá el elemento de la orilla izquierda y guardará la nueva lista en **OS1** y luego insertará el elemento en la orilla derecha y guardará la lista modificada en **OS2** . En **OS1** y en **OS2** tenemos el estado actual de las listas.

Ahora si, soy capaz de transportar un elemento de una orilla a otra, pero tenemos una serie de restricciones. No podemos dejar al lobo con la oveja porque se la comería y lo mismo con la oveja y la col. No obstante las condiciones anteriores se anulan siempre que se encuentre el pastor. Es decir que los movimientos que podemos hacer están condicionados. Podemos realizar movimientos de una orilla a otra independientemente de a qué orilla se muevan los elementos siempre que tengamos en cuenta estas condiciones. Entonces ¿Cómo comprobamos en qué lista están los elementos?. Con la función *perteneceLista* que a su vez utiliza pertenece. Al comprobar en qué lista están los elementos podemos evitar caer en alguna de las restricciones.

```
pertenece(X, [X|_]).
pertenece(X, [_|Y]) :- pertenece(X,Y).
perteneceLista([], _).
perteneceLista([H|T], L):- pertenece(H,L), perteneceLista(T,L).
```

La función *pertenece* comprueba si un elementos se encuentra en la lista. La primera sentencia comprueba si el elemento es la cabeza, sino entra por la segunda sentencia y vuelve a llamar a pertenece quitándole la cabeza y enviando la cola. Es decir, vamos desplazando los elementos a la cabeza para comprobarlos uno a uno.

La función *perteneceLista* funciona de manera similar. *PerteneceLista* comprueba si una lista de elementos se encuentra dentro de otra lista de elementos. La primera sentencia nos indica que si comprobamos todos los elementos de la lista izquierda con éxito, nos dirá que sí pertenecen, mientras tanto en la segunda sentencia llamará al pertenece pasándole como elemento la cabeza de la lista y así sucesivamente hasta que los hayamos comprobado todos.

Una vez podemos comprobar si los elementos están en la orilla o no podemos implementar los diferentes movimientos condicionados por las restricciones. Las condiciones se llamarán mover y tendrán una serie de requisitos que cumplir para poder transportar(**viajar**) un elemento de una orilla a otra.

```
/* Condiciiones de movimientos */
/*Izquierda a derecha */

mover(OE1,OE2,OS1,OS2):- pertenece(pastor,OE1), pertenece(oveja,OE1)
   , viajar(OE1,OE2,oveja,Q,P), viajar(Q,P,pastor,OS1,OS2).
mover(OE1,OE2,OS1,OS2):-pertenece(pastor,OE1), pertenece(col,OE1),
    viajar(OE1,OE2,col,Q,P), viajar(Q,P,pastor,OS1,OS2),
    not(perteneceLista([oveja,lobo],OE1)).
mover(OE1,OE2,OS1,OS2):-pertenece(pastor,OE1), pertenece(lobo,OE1),
    viajar(OE1,OE2,lobo,Q,P), viajar(Q,P,pastor,OS1,OS2),
    not(perteneceLista([oveja,col],OE1)).
mover(OE1,OE2,OS1,OS2):-viajar(OE1,OE2,pastor,OS1,OS2),
    not(perteneceLista([oveja,col],OE1)) | not(perteneceLista([oveja,lobo],OE1)).
```

Vamos a analizar los movimientos de la orilla izquierda a la derecha. *Mover* tiene como parámetros las dos orillas antes de realizar el movimiento (**OE1,OE2**) y otros dos parámetros donde se almacenan estas orillas una vez realizado el movimiento (**OS1, OS2**). Analicemos *mover* en concreto con uno de los ejemplos.

```
mover(OE1,OE2,OS1,OS2):- pertenece(pastor,OE1), pertenece(oveja,OE1)
   , viajar(OE1,OE2,oveja,Q,P), viajar(Q,P,pastor,OS1,OS2).
```

Para poder hacer el movimiento de la oveja, comprueba que tanto la oveja como el pastor están en esa lista. Si están en la lista mueve a cada uno de los elementos con la función *viajar*. La oveja no tiene restricciones aparte de estar en la propia lista con el pastor para moverse. Aunque deje al lobo y la col solos no ocurriría nada. ¿Cómo sería con el lobo?.

```
mover(OE1,OE2,OS1,OS2):-pertenece(pastor,OE1), pertenece(lobo,OE1),
    viajar(OE1,OE2,lobo,Q,P), viajar(Q,P,pastor,OS1,OS2),
    not(perteneceLista([oveja,col],OE1)).
```

El lobo tiene una serie de restricciones aparte de las comentadas anteriormente. El lobo antes de moverse tienen que comprobar si nos pertenecen la oveja y la col a la lista donde está el actualmente. Si no pertenecen, es decir no están en la misma lista que el lobo, el lobo se podría mover con el pastor sin dejar solos a la oveja y a la col. ¿Por qué no usamos *pertenece* y comprobamos elementos a elemento?. No podemos usar *pertenece* porque puede que uno de los elementos si se encuentre en la lista y el otro no. Esto no afecta al problema, mientras no estén juntos sin el pastor no hay ningún problema. El movimiento de la col es muy similar y lo mismo con el pastor. Lo único a tener en cuenta es que el pastor no puede moverse solo dejando a la oveja y la col juntas o al lobo y la oveja juntos, mientras que la col no puede moverse dejando al lobo y la col solos. Los movimientos que vimos en la figura son de izquierda a derecha, los movimientos de derecha a izquierda son exactamente igual cambiandoles las orillas.

```
/* Derecha a izquierda*/

mover(OE1,OE2,OS1,OS2):- pertenece(pastor,OE2), pertenece(oveja,OE2)
   , viajar(OE2,OE1,oveja,Q,P), viajar(Q,P,pastor,OS2,OS1).
mover(OE1,OE2,OS1,OS2):-pertenece(pastor,OE2), pertenece(col,OE2),
    viajar(OE2,OE1,col,Q,P), viajar(Q,P,pastor,OS2,OS1),
    not(perteneceLista([oveja,lobo],OE2)).
mover(OE1,OE2,OS1,OS2):-pertenece(pastor,OE2), pertenece(lobo,OE1),
    viajar(OE2,OE1,lobo,Q,P), viajar(Q,P,pastor,OS2,OS1),
    not(perteneceLista([oveja,col],OE2)).
mover(OE1,OE2,OS1,OS2):-viajar(OE2,OE1,pastor,OS2,OS1),
    not(perteneceLista([oveja,col],OE2)) | not(perteneceLista([oveja,lobo],OE2)).
```

Recapitulemos un poco que he hecho hasta el momento. Tenemos una función *viajar* que lo que hace es extraer de una lista e insertar en otra. Con esta función podemos transportar elementos de una orilla a otra. Pero estos movimientos están condicionados por quiénes están en las orillas, por lo que necesitamos un método para comprobar quienes están en ellas. El método *pertenece* y *perteneceLista*. Por último he creado las condiciones para hacer movimientos seguros sin caer en las restricciones. ¿Qué falta?. El problema se tiene que estar ejecutando hasta que haga los movimientos necesarios para la solución, por lo que necesitamos un método que se encargue de ir haciendo los movimientos correctamente. El método es *sol*. Antes de explicar sol hay que tener en cuenta que un movimiento seguro podría repetirse infinitas veces por lo que nunca llegaríamos a la solución. ¿Cómo podemos solucionarlo? Con un acumulador de movimientos. Podemos ir analizando qué movimientos hemos realizado y si lo hemos realizado buscar otro y así consecutivamente hasta llegar al resultado.

```
sol([],_,ACC,ACC).
sol(OE1, OE2, ACC, RESULT):- mover(OE1,OE2,OS1,OS2), not(pertenece([OS1,OS2],ACC)),
    append(ACC,[[OS1,OS2]],ACC2),sol(OS1,OS2,ACC2,RESULT).
```

Los parámetros que tiene solución son los siguientes: la lista de la orilla izquierda (**OE1**), la lista de la orilla derecha(**OE2**), el acumulador(**ACC**), y el resultado (**RESULT**) con todos los movimientos realizados.

La primera sentencia de sol nos dice que cuando hayamos movido todos los elementos de la orilla izquierda a la orilla derecha habremos acabado el problema y nos devolverá el acumulador con todos los movimientos realizados. La segunda sentencia busca un movimiento que pueda realizar, comprueba si ya ha realizado este movimiento, sino añade al acumulador con append el nuevo movimiento y hace una llamada recursiva a sol para hacer otro movimiento hasta llegar a la solución. ¿Qué hace el append?

```
append([],L,L).
append([H|T],L2,[H|L3]):-append(T,L2,L3).
```

Pues el append concatena una lista con otra así podemos ir concatenando las distintas lista de movimientos e ir aumentando el número de movimientos en el acumulador.

Entonces *sol* nunca realizará movimientos que ya hayamos realizado por lo que no entraremos en bucle infinito sino que acabará al encontrar la solución.

4.2. Dificultades

A lo largo de la práctica tuve una buena cantidad de dificultades. La mayor dificultad que tuvimos fue asimilar que en prolog no se pueden revalorizar las variables. Una vez solucionada esa dificultad también tuve bastantes problemas con las condiciones. Por ejemplo como comente arriba, no podemos utilizar *pertenece* por separado para comprobar si está la oveja y la col juntas o el lobo y la oveja juntas en una lista porque podría una si estar y solo nos afecta si están las dos juntas. Por este motivo el programa se nos quedaba en tres movimientos hasta que lo solucioné con el *perteneceLista*. La mayoría de problemas que tuve fue por revalorizar variables y por no tener algunos conceptos que fuimos desarrollando en las clase práctica mientras hacíamos la práctica.

5. Conclusión

En conclusión, este programa tiene una función viajar que permite transportar elementos de una orilla a otra. Como las orillas están representadas por lista lo que hace es extraer de una lista e insertar en otra. Estos movimientos están condicionados por lo que antes de transportar los elementos con la función viajar, en mover se comprueba que cumpla las restricciones. El programa consta también con una función sol que elige los movimientos hasta llegar al resultado. Para evitar que los movimientos se repitan y entremos en bucle infinito, la función sol consta de un acumulador donde se van almacenando todos los movimientos que vamos realizando y así podemos comprobar si se repiten. Una vez llegada a la solución nos muestra todos los pasos que hemos realizado para llegar a la solución.

La sentencia para ejecutar el código es esta:

sol([pastor,col,lobo,oveja],[],[[[pastor,col,lobo,oveja],[]]],R).

El acumulador contiene el movimiento inicial para que se contemple cuando se muestre el resultado.

6. Opinión personal

La práctica ha supuesto un reto muy divertido, por mi parte he sufrido sacándola pero también me he divertido haciéndola. Además de que ha servido como motivación para afrontar las demás prácticas que nos quedan de otras asignaturas. Aunque me haya costado mucho y con ayuda de los vídeos de la profesora María Belén Melián Batista y en páginas de Google o Youtube, estoy satisfecho por el esfuerzo que he realizado para sacarla.

En conclusión, esta práctica ha sido un reto personal que me ha ayudado a superarme.