LABORATORIOS DE INTELIGENCIA ARTIFICIAL
CURSO 2017–2018

PRÁCTICA 3: PROLOG

Fecha de publicación: 2018/04/05

Fechas de entrega: grupos jueves: miércoles 2018/04/18, 23:55
grupos viernes: jueves 2018/04/19, 23:55

Planificación: Ejercicios para la 1.ª semana: 1, 2, 3 y 4.
Ejercicios para la 2.ª semana: 5, 6, 7 y 8.

Entrega: Para la entrega en formato electrónico, se creará un archivo zip que contenga todo el material de la entrega, cuyo nombre, todo él en minúsculas y sin acentos, tildes, o caracteres especiales, tendrá la siguiente estructura:

[gggg]_p3_[mm]_[apellido1]_[apellido2].zip

donde

- [gggg] : Número de grupo de la pareja: 2301, 2311, 2312, 2313 2361, 2362 o 2363
- [mm] : Número de orden de la pareja con dos dígitos (posibles valores: 01, 02, 03,...)
- [apellido1]: Primer apellido del miembro1 de la pareja
- [apellido2] : Primer apellido del miembro2 de la pareja

Los miembros de la pareja aparecen en orden alfabético. Ejemplos :

- 2311_p3_01_delval_sanchezmontanyes.zip (práct. 3 de la pareja 01 en el grupo de prácticas 2311)
- 2362_p3_18_salabert_suarez.zip (práctica 3 de la pareja 18 en el grupo de prácticas 2362)

Contenido del fichero comprimido:

- El archivo de código Prolog: [gggg]_p3_[mm].pl
- El archivo: [gggg]_p3_[mm]_readme.txt [OPCIONAL]
- Los archivos de la memoria (nombre siempre en minúsculas): [gggg]_p3_[mm]_memoria.txt o [gggg]_p3_[mm]_memoria.pdf

---

EJERCICIO 1 (1 punto). Considera la siguiente implementación del predicado `pertenece(X, L)` que comprueba si un elemento está en una lista:

```
pertenece(X, [X|_]).
pertenece(X, [_|Rs]) :- pertenece(X, Rs).
```

El predicado se satisface tantas veces como elementos repetidos contenga la lista.

```
?- pertenece(1, [2, 1, 3, 1]).
true ;
true ;
false.
```

El predicado funciona tanto si `X` como `L` no están instanciadas y genera por *backtracking* y en orden todas las soluciones:

```
?- pertenece(X, [2,1,3,1]).
X = 2 ;
X = 1 ;
X = 3 ;
X = 1 ;
false.

?- pertenece(1, L).
L = [1|_G12270] ;
L = [_G12269, 1|_G12273] ;
L = [_G12269, _G12272, 1|_G12276] ;
L = [_G12269, _G12272, _G12275, 1|_G12279] ;
L = [_G12269, _G12272, _G12275, _G12278, 1|_G12282]
```

Utiliza los predicados trace y notrace para trazar la ejecución de los objetivos anteriores, consulta el tutorial de Prolog para ver cómo se usan.

Una vez que hayas entendido el funcionamiento de estos predicados implementa un nuevo predicado `pertenece_m(X, L)` que comprueba si el elemento está en la lista o en algunas de sus sublistas.

```
?- pertenece_m(X, [2,[1,3],[1,[4,5]]]).
X = 2 ;
X = 1 ;
X = 3 ;
X = 1 ;
X = 4 ;
X = 5 ;
false.
```

Ayuda: una forma de imponer que una variable `Y` no es una lista es `Y \= [_|_]`

EJERCICIO 2 (1 punto). Implementa el predicado `invierte(L, R)` que se satisface cuando `R` contiene los elementos de `L` en orden inverso. Utiliza el predicado `concatena/3` (/n: indica n argumentos):

```
concatena([], L, L).
concatena([X|L1], L2, [X|L3]) :-
    concatena(L1, L2, L3).
```

que se satisface cuando su tercer argumento es el resultado de concatenar las dos listas que se dan como primer y segundo argumento.

Ejemplos:

```
?- concatena([], [1, 2, 3], L).
L = [1, 2, 3].

?- concatena([1, 2, 3], [4, 5], L).
```

```
L = [1, 2, 3, 4, 5].

?- invierte([1, 2], L).
L = [2, 1]

?- invierte([], L).
L = []

?- invierte([1, 2], L).
L = [2, 1]
```

EJERCICIO 3 (1 punto). Implementar el predicado `insert(X-P, L, R)` que inserte un par de elementos (`X-P`) en una lista de pares ordenados (`L`) en una posición (`P`), desplazando el resto de elementos, en la lista (`R`). Se considera que la primera posición de una lista es la 1. Como en el anterior ejercicio, utiliza el predicado `concatena/3`:

```
?- insert([a-6],[], X).
X = [a-6].

?- insert([a-6],[p-0], X).
X = [p-0, a-6].

?- insert([a-6],[p-0, g-7], X).
X = [p-0, a-6, g-7],
false.

?- insert([a-6],[p-0, g-7, t-2], X).
X = [p-0, a-6, g-7, t-2],
false.
```

Ejemplos:

EJERCICIO 4.1 (0.5 puntos). Implementar el predicado `elem_count(X, L, Xn)` que se satisface cuando el elemento (`X`) aparece (`Xn`) veces en la lista (`L`).

```
?- elem_count(b,[b,a,b,a,b],Xn).
Xn = 3,
false.

?- elem_count(a,[b,a,b,a,b],Xn).
Xn = 2,
false.
```

EJERCICIO 4.2 (0.5 puntos). Implementar el predicado `list_count(L1, L2, L3)` que se satisface cuando la lista (`L3`) contiene las ocurrencias de los elementos de (`L1`) en (`L2`) en forma de par. Ejemplos:

```
?- list_count([b],[b,a,b,a,b],Xn).
Xn = [b-3]
False

?- list_count([b,a],[b,a,b,a,b],Xn).
Xn = [b-3, a-2]
False

?- list_count([b,a,c],[b,a,b,a,b],Xn).
Xn = [b-3, a-2, c-0]
false
```

EJERCICIO 5 (1 punto). Implementar el predicado `sort_list(L1, L2)` que se satisface cuando la lista (`L2`) contiene los pares de elementos de la lista (`L1`) en orden. Ejemplos:

```
?-sort_list([p-0, a-6, g-7, t-2], X).
X = [p-0, t-2, a-6, g-7]
false

?-sort_list([p-0, a-6, g-7, p-9, t-2], X).
X = [p-0, t-2, a-6, g-7, p-9]
false

?-sort_list([p-0, a-6, g-7, p-9, t-2, 9-99], X).
X = [p-0, t-2, a-6, g-7, p-9, 9-99]
false
```

Un árbol de Huffman es una estructura usada para **codificar y comprimir información**. El objetivo es codificar un texto formado por símbolos utilizando códigos binarios. El esquema de codificación utiliza la frecuencia que aparece cada uno de los símbolos en un texto para minimizar el número de bits necesario para almacenar la información. En concreto,los símbolos más frecuentes en el texto están codificados con bloques de bits más cortos,. A modo de ejemplo, la cadena en ASCII,

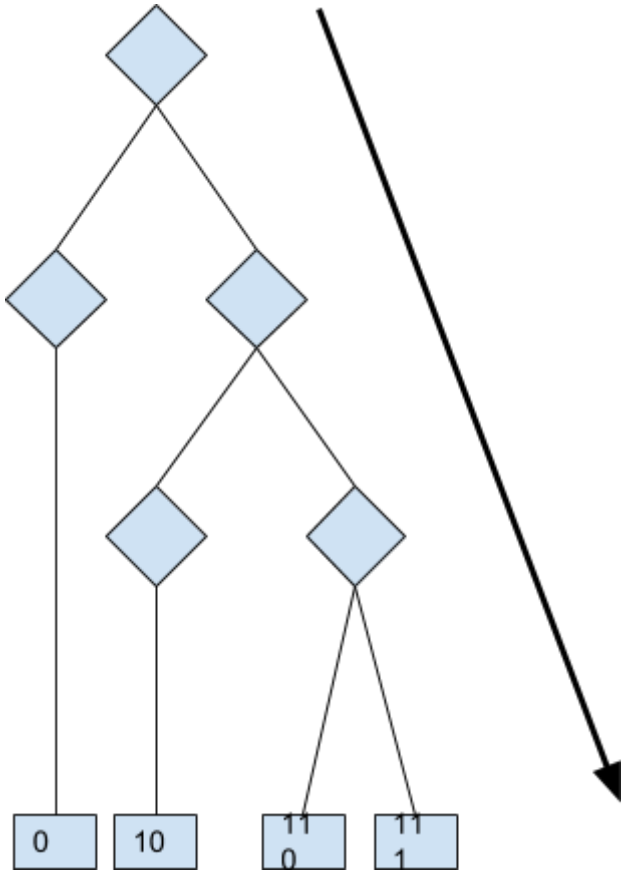
```
AAAADAAACCCAAAAAAABAAAAABAAA
```

usa para almacenar cada caracter 1 Byte. Sin embargo, podemos reducir el espacio ocupado por la cadena. De forma simple podemos expresarlo como:

A	0
B	111
C	10
D	110

De este modo, utilizando los códigos ASCII por los códigos de bloques de bits especificados en esta tabla, es posible reducir la longitud de la cadena sin perder información. Es importante observar que ningún código es prefijo de otro código (por ejemplo, solo el código del carácter 'A', el más frecuente en la cadena, empieza por '0'. De forma análoga, solo el código del carácter 'C' comienza por '10', etc.), por lo que no hay ambigüedad al descodificar.
En esta práctica implementaremos una versión simplificada de este tipo de estructura.

Para ello insertaremos el elemento con mayor probabilidad a la izquierda del nodo raíz, y seguiremos con este mismo proceso expandiendo el árbol por la derecha. Cuando solo queden 2 elementos dejaremos de expandir el árbol y terminaremos como se muestra en la figura.



EJERCICIO 6 (1 punto). Implementa el predicado `build_tree(List, Tree)` que transforma una lista de pares de elementos ordenados en una versión simplificada de un árbol de Huffman. Para representar árboles usaremos las funciones `tree(Info, Left, Right)` y `nil`. También usaremos el predicado `concatena/3`. Ejemplo:

```
?-build_tree([p-0, a-6, g-7, p-9, t-2, 9-99], X).
X = tree(1, tree(p, nil, nil), tree(1, tree(a, nil, nil), tree(1, tree(g, nil, nil),
tree(1, tree(p, nil, nil), tree(1, tree(t, nil, nil), tree(9, nil, nil))))))
false

?-build_tree([p-55, a-6, g-7, p-9, t-2, 9-99], X).
X = tree(1, tree(p, nil, nil), tree(1, tree(a, nil, nil), tree(1, tree(g, nil, nil),
tree(1, tree(p, nil, nil), tree(1, tree(t, nil, nil), tree(9, nil, nil))))))
False
```

```
?-build_tree([p-55, a-6, g-2, p-1], X).
X = tree(1, tree(p, nil, nil), tree(1, tree(a, nil, nil), tree(1, tree(g, nil, nil),
tree(p, nil, nil))))
False

?-build_tree([a-11, b-6, c-2, d-1], X).
X = tree(1, tree(a, nil, nil), tree(1, tree(b, nil, nil), tree(1, tree(c, nil, nil),
tree(d, nil, nil))))
```

Los nodos hoja del árbol se corresponden con los elementos de la lista ordenada y almacenan el elemento en el campo `Info`, mientras que los nodos intermedios siempre almacenan un 1.

EJERCICIO 7.1 (1 punto). Implementar el predicado `encode_elem(X1, X2, Tree)` que codifica el elemento (`X1`) en (`X2`) basándose en la estructura del árbol (`Tree`). Así tomando el árbol del ejemplo anterior

```
?-build_tree([a-11, b-6, c-2, d-1], X).
X = tree(1, tree(a, nil, nil), tree(1, tree(b, nil, nil), tree(1, tree(c, nil, nil),
tree(d, nil, nil))))
```

Tenemos los siguientes ejemplos:

```
?- encode_elem(a, X, tree(1, tree(a, nil, nil), tree(1, tree(b, nil, nil), tree(1,
tree(c, nil, nil), tree(d, nil, nil))))).
X = [0]
false

?- encode_elem(b, X, tree(1, tree(a, nil, nil), tree(1, tree(b, nil, nil), tree(1,
tree(c, nil, nil), tree(d, nil, nil))))).
X = [1, 0]
false

?- encode_elem(c, X, tree(1, tree(a, nil, nil), tree(1, tree(b, nil, nil), tree(1,
tree(c, nil, nil), tree(d, nil, nil))))).
X = [1, 1, 0]
false

?- encode_elem(d, X, tree(1, tree(a, nil, nil), tree(1, tree(b, nil, nil), tree(1,
tree(c, nil, nil), tree(d, nil, nil))))).
X = [1, 1, 1]
false
```

EJERCICIO 7.2 (1 punto). Implementar el predicado `encode_list(L1, L2, Tree)` que codifica la lista (`L1`) en (`L2`) siguiendo la estructura del árbol (`Tree`). Ejemplos:

```
?- encode_list([a], X, tree(1, tree(a, nil, nil), tree(1, tree(b, nil, nil), tree(1,
tree(c, nil, nil), tree(d, nil, nil))))).
```

```
X = [[0]]
false

?- encode_list([a,a], X, tree(1, tree(a, nil, nil), tree(1, tree(b, nil, nil), tree(1,
tree(c, nil, nil), tree(d, nil, nil))))).
X = [[0], [0]]
false

?- encode_list([a,d,a], X, tree(1, tree(a, nil, nil), tree(1, tree(b, nil, nil),
tree(1, tree(c, nil, nil), tree(d, nil, nil))))).
X = [[0], [1, 1, 1], [0]]
false

?- encode_list([a,d,a,q], X, tree(1, tree(a, nil, nil), tree(1, tree(b, nil, nil),
tree(1, tree(c, nil, nil), tree(d, nil, nil))))).
false.
```

EJERCICIO 8 (2 puntos). Implementar el predicado `encode(L1, L2)` que codifica la lista (`L1`) en (`L2`). Para ello haced uso del predicado `dictionary`

```
dictionary([a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z]).
```

Ejemplos:

```
?- encode([i,n,t,e,l,i,g,e,n,c,i,a,a,r,t,i,f,i,c,i,a,l],X).
X = [[0], [1, 1, 1, 0], [1, 1, 0], [1, 1, 1, 1, 1, 0], [1, 1, 1, 1, 0], [0], [1, 1, 1,
1, 1, 1, 1, 1, 0], [1, 1, 1, 1, 1, 0], [1, 1, 1, 0], [1, 1, 1, 1, 1, 1, 0], [0], [1,
0], [1, 0], [1, 1, 1, 1, 1, 1, 1, 0], [1, 1, 0], [0], [1, 1, 1, 1, 1, 1, 1, 1, 1, 0],
[0], [1, 1, 1, 1, 1, 1, 0], [0], [1, 0], [1, 1, 1, 1, 0]]
False

?- encode([i,a],X).
X = [[0], [1, 0]]
False

?- encode([i,2,a],X).
false
```