Trabajo Práctico 2: Programación Lógica

Paradigmas de Lenguajes de Programación — 2do cuat. 2015

Fecha de entrega: 3 de noviembre

1. Introducción

El problema a resolver en el trabajo consiste en generar todas las maneras posibles de cubrir una longitud determinada con piezas de diferentes tamaños, donde, además, la cantidad de cada una es limitada.[1]

Se espera que se exprese en Prolog de forma elemental, declarativa y clara algunas variantes de este problema según la guía de trabajo, comenzando por una versión *naïve*, para, luego, incorporar un enfoque similar a la *programación dinámica* que permita reducir el tiempo de búsqueda de las soluciones. En este sentido, se espera también que se mida y observe efectivamente cómo esta incorporación acorta el tiempo de ejecución del programa.

Representación utilizada

Las piezas disponibles se representan con una lista donde se indica tamaño y cantidad disponible de cada una, y se puede suponer que están ordenadas por tamaño ascendente: `[pieza(t1, c1), pieza(t2, c2), ..., pieza(tn, cn)]`.

Por otro lado, las posibles soluciones se representan con listas de elementos que representan el tamaño de cada pieza. Por ejemplo, se podría cubrir una longitud de tamaño 21 con `[2,2,2,2,2,1,2,2,2,2,2]`, `[1,7,4,4,4,1]`, `[10,10,1]` y `[21]`.

2. Guía de trabajo

Calentando motores

Comencemos programando los siguientes predicados, que serán de utilidad para el trabajo:

1. `listaNats(+LInf,+LSup,?Nats)`, que unifica la lista `Nats` con los naturales en el rango `[LInf, LSup]`, o una lista vacía si `LSup` $<$ `LInf`.
2. `nPiezasDeCada(+Cant, +Tamaños, -Piezas)`, que instancia a `Piezas` con una lista que contiene una cantidad `Cant` de cada tamaño en la lista `Tamaños`. Así, se pueden generar listas de piezas como las siguientes:

```
ejemplo(e1, P) :- listaNats( 1, 3, D_1_3), nPiezasDeCada( 2, D_1_3, P_1_3),
                  listaNats( 4, 6, D_4_6), nPiezasDeCada( 5, D_4_6, P_4_6), append(P_1_3, P_4_6, P).
ejemplo(e2, P) :- listaNats( 1, 20, D), nPiezasDeCada( 5, D, P).
...
?- ejemplo(e1,L).
L = [pieza(1, 2), pieza(2, 2), pieza(3, 2), pieza(4, 5), pieza(5, 5), pieza(6, 5)]

?- ejemplo(e2,L).
L = [pieza(1,5), pieza(2,5), pieza(3,5), pieza(4,5), pieza(5,5), pieza(6,5), pieza(7,5),
     pieza(8,5), pieza(9,5), pieza(10,5), pieza(11,5), pieza(12,5), pieza(13,5), pieza(14,5),
     pieza(15,5), pieza(16,5), pieza(17,5), pieza(18,5), pieza(19,5), pieza(20,5)]
```

[1] Esto es similar al *problema del vuelto*, donde se quiere devolver una determinada cantidad de dinero disponiendo de una cantidad limitada de monedas o billetes de cada denominación. Formalmente, a partir de un conjunto $\{(m_1, c_1), \ldots, (m_n, c_n)\}$ y un valor V, se quieren encontrar todos los $I \subseteq \{1 \ldots n\}$ tales que $\sum_{i \in I} m_i k_i = V$, con $k_i \leq c_i \forall i \in I$.

3. `resumenPiezas(+SecPiezas, -Piezas)`, que permite instanciar `Piezas` con la lista de piezas, con su cantidad, incluidas en `SecPiezas`. Por ejemplo,

```
?- resumenPiezas([2,2,1,2,3,10,1], L).
L = [pieza(1,2), pieza(2,3), pieza(3,1), pieza(10,1)].
```

Enfoque *naïve*

Para una primera versión pide implementar los siguientes predicados, para lo cual se pretende encontrar las soluciones mediante la técnica de *generate and test*:

4. `generar(+Total,+Piezas,-Solución)`, donde `Solución` representa una lista de piezas cuyos valores suman `Total`. Aquí no se pide controlar que la cantidad de cada pieza esté acorde con la disponibilidad.

5. `cumpleLímite(+Piezas,+Solución)` será verdadero cuando las cantidades por pieza utilizadas en `Solución` no excedan las disponibles indicadas en `Piezas`.

6. `construir1(+Total,+Piezas,-Solución)`, donde `Solución` representa una lista de piezas cuyos valores suman `Total` y, además, las cantidades utilizadas de cada pieza no exceden los declarados en `Piezas`.

Enfoque *dinámico*

Ahora se desea conseguir las mismas soluciones que en el caso anterior, pero sin que la recursión resulte en la repetición de cálculos. Para ello se pide ver el resumen en la sección 5.1, y desarrollar los siguientes predicados:

7. `construir2(+Total,+Piezas,-Solución)`, cuyo comportamiento es idéntico a `construir1/3`, pero que utiliza definiciones dinámicas para persistir los cálculos auxiliares realizados y evitar repetirlos. No se espera que las soluciones aparezcan en el mismo orden entre `construir1/3` y `construir2/3`, pero sí que sean las mismas.

   Se sugiere definir una solución pensando en un equema de *programación dinámica*, separando el problema en casos más chicos que van a usarse varias veces a lo largo de la búsqueda de una solución del problema original. Luego, a la hora de resolver un subproblema se podrá verificar si ya se había resuelto en el pasado y aprovechar así una enumeración explícita de soluciones sin repetir operaciones. Por si tienen problemas, tienen una ayuda extra en la sección 5.2.

Comparación de resultados y tiempos

La idea de sección es comparar los resultados obtenidos mediante los dos enfoques anteriores. Para ello, se pide implementar los siguientes predicados:

8. `todosConstruir1(+Total, +Piezas, -Soluciones, -N)`, donde `Soluciones` representa una lista con todas las soluciones de longitud `Total` obtenidas con `construir1/3`, y `N` indica la cantidad total de soluciones.

9. `todosConstruir2(+Total, +Piezas, -Soluciones, -N)`, ídem al anterior predicado, pero utilizando `construir2/3`.

Se deberá verificar que las soluciones encontradas son las mismas y, además, se deberá medir el tiempo que toma cada una. Los resultados deberán incorporarse como comentarios en el código. Para conocer el tiempo de ejecución, pueden correr las consultas utilizando el predicado `time/1`.

Patrones

Ahora se pretende encontrar soluciones que respeten un cierto *patrón*, para lo que deberá implementarse el siguiente predicado:

10. `construirConPatron(+Total, +Piezas, ?Patrón, -Solución)` será verdadero cuando `Solución` sea una solución factible en los términos definidos anteriormente y, además, sus piezas respeten el patrón indicado en `Patrón`.

    Se sugiere definir un predicado `tienePatrón(+Patrón, ?Lista)` que decida si `Lista` presenta el `Patrón` especificado.
    Un patrón está determinado por una lista con variables o literales. Consideraremos que una lista respeta un patrón cuando la misma variable del patrón se corresponde siempre con el mismo elemento de la lista y además coinciden los literales de ambos. En el caso en que la longitud del patrón sea menor que la longitud de la lista, éste deberá repetirse una cantidad entera de veces.

```
?- tienePatrón([A, A], [1,1,1,1]).
A = 1 .
?- tienePatrón([B, B], [1,1,1]).
false.
?- tienePatrón([A, B], [2,1,2,1]).
A = 2, B = 1 .
```

```
?- tienePatrón([A, B], [2,1,4,5]).
false.
?- tienePatrón([A, 1, B, 2, C], [2,1,3,2,7]).
A = 2, B = 3, C = 7 .
```

3. Condiciones de aprobación

El principal objetivo de este trabajo es evaluar el correcto uso del lenguaje PROLOG de forma declarativa para resolver el problema planteado. El código debe estar *adecuadamente* comentado (es decir, los comentarios que escriban deben facilitar la lectura de los predicados, y no ser una traducción al castellano del código). También se debe explicitar cuáles de los argumentos de los predicados auxiliares deben estar instanciados usando `+`, `-` y `?`.

La entrega debe incluir casos de prueba en los que se ejecute al menos una vez cada predicado definido, además de las mediciones de tiempo en los ejercicios que así lo requieran.

En el caso de los predicados que utilicen la técnica de *generate and test*, deberán indicarlo en los comentarios.

4. Pautas de Entrega

Se debe entregar el código impreso con la implementación de los predicados pedidos. Cada predicado debe contar con un comentario donde se explique su funcionamiento. Cada predicado asociado a los ejercicios debe contar con ejemplos que muestren que exhibe la funcionalidad solicitada. Además, se debe enviar un e-mail conteniendo el código fuente en Prolog a la dirección plp-docentes@dc.uba.ar. Dicho mail debe cumplir con el siguiente formato:

- El título debe ser `[PLP;TP-PL]` seguido inmediatamente del nombre del grupo.
- El código Prolog debe acompañar el e-mail y lo debe hacer en forma de archivo adjunto con nombre `tp2.pl`.

El código debe poder ser ejecutado en SWI-Prolog. No es necesario entregar un informe sobre el trabajo, alcanza con que el código esté adecuadamente comentado. Los objetivos a evaluar en la implementación de los predicados son:

- corrección,
- declaratividad,
- reutilización de predicados previamente definidos
- Uso de unificación, backtracking, generate and test y reversibilidad de los predicados.
- **Importante**: salvo donde se indique lo contrario, los predicados no deben instanciar soluciones repetidas. Vale aclarar que no es necesario filtrar las soluciones repetidas si la repetición proviene de las características de la entrada.

Importante: se admitirá un único envío, sin excepción alguna. Por favor planifiquen el trabajo para llegar a tiempo con la entrega.

5. Referencias y sugerencias

Como referencia se recomienda la bibliografía incluída en el sitio de la materia (ver sección *Bibliografía → Programación Lógica*).

Se recomienda que utilicen los predicados ISO y los de SWI-Prolog ya disponibles, siempre que sea posible. Recomendamos especialmente examinar los predicados y metapredicados que figuran en la sección *Cosas útiles* de la página de la materia. Pueden hallar la descripción de los mismos en la ayuda de `SWI-Prolog` (a la que acceden con el predicado `help`). También se puede acceder a la documentación online de SWI-Prolog.

5.1. Manejo de predicados dinámicos

Prolog permite agregar hechos a la base de conocimientos de un programa en tiempo de ejecución mediante la utilización de *predicados dinámicos*. Esto se debe indicar en el código mediante `dynamic`, lo cual habilita a utilizar `asserta/1` y `assertz/1` para agregar hechos al comienzo o al final de la base, respectivamente. Además, existe `retractall/1` que elimina todas las definiciones dinámicas hasta el momento del predicado que recibe. Por ejemplo:

```
%indicar que se van a agregar a la base predicados de la forma hechoNotable(X, Y).
:- dynamic hechoNotable/2.

%elegirHecho(+A, +B, -C)
elegirHecho(A, B, C) :-
    retractall(hechoNotable(_,_)),
    %borra todas las definiciones dinámicas que unifiquen con hechoNotable(_,_)
    generarHechos(A, B),
    elegirUno(A, B, C).
```

```
%generarHecho(+A, +B)
generarHechos(A, B):-
    ...,
    %X e Y surgieron de las cláusulas anteriores
    asserta(hechoNotable(X, Y)),
    %ahora hechoNotable(X, Y) está AL COMIENZO de la base de conocimiento
    ... .

%elegirUno(+A, +B, -C)
elegirUno(A, B, C) :-
    ...,
    ... hechoNotable(B, D),
    %consulta la base de conocimentos donde deberá haber definiciones de hechoNotable
    ...
    % en algún momento se instanciará C
```

Algunas observaciones:

- Utilizamos `retractall` para borrar definiciones que hayan surgido de ejecuciones anteriores y que ya no tengan sentido.
- Dado que es importante el orden en el que aparecen las definiciones, se debe ser cuidadoso al elegir entre `asserta` y `assertz`.
- Debemos asegurarnos que antes de que se alcance una cláusula que utiliza un predicado dinámico existan definiciones de éste en la base de conocimientos.

Para más información, pueden explorar las páginas de documentación citadas y no dejarse amedrentar por este resumen.

5.2. Cómo aprovechar resultados anteriores

Se da una forma de aprovechar las listas calculadas anteriormente (usar sólo si se necesita).

Soluciones intermedias

Se puede pensar a las listas que suman T, usando ciertas piezas P, como soluciones intermedias al problema en el cual se buscan las listas que suman T, usando de ciertas piezas P, pero sólo aquellas que son a lo sumo de largo K ($\overline{\langle T, P, K\rangle}$).
Entonces, el conjunto de listas para $\langle T, P, K\rangle$, sacando el caso trivial donde no es necesario poner piezas ($T = 0$), en P habría una pieza de largo L que es la más larga por debajo de K. Así, puede suceder que

a) la pieza de largo L no aparece, estas serían las listas para $\langle T, P, L-1\rangle$,

b) la pieza de largo L aparece al menos una vez.

En el caso b) a la izquierda de la primera aparición de L van a haber ciertas piezas que suman T_1 y son todas menores que L. O sea, alguna combinación que pertenece a $\langle T_1, P, L-1\rangle$. A la derecha de la primera aparición de L, van a aparecer ciertas piezas que suman T_2 (notar que $T = T_1 + L + T_2$) dentro de las cuales pueden aparecer otras de largo L, pero no más largas que K, con lo cual pertenecen a $\langle T_2, P, L\rangle$.