Laboratorio de Lenguajes de Programación

USB / CI-3661 / Sep-Dic 2016

(Programación Lógica – 25 puntos)

Arre caballito…

¿Podemos ubicarlos? (5 puntos)

Escriba el predicado `arre/3`, que será invocado unificando su primer argumento con un número entero positivo `N`, y que debe triunfar unificando su segundo argumento con el número *máximo* de caballos que pueden colocarse en un tablero de ajedrez de lado `N` sin que se ataquen entre si, y su tercer argumento con una lista que indique las posiciones en las cuales se ubican los caballos.

La lista resultante debe tener la forma

```
[k(R0,C0),k(R1,C1),...]
```

donde `R0` y `C0` son los números enteros entre `1` y `N` indicando la fila y columna, respectivamente, en que está ubicado el primer caballo, y así sucesivamente. No importa el orden en que aparezcan en la lista, lo que importa es que aparezcan todas las posiciones y que no haya repetidos.

Ejemplos (triviales) de invocación

```
?- arre(1,N,L).
N = 1
L = [k(1,1)]

?- arre(2,N,L).
N = 4
L = [k(1,1),k(1,2),k(2,1),k(2,2)]
```

Un ejemplo más interesante sería

```
?- arre(3,N,L).
N = 5
L = [k(1,1),k(1,3),k(2,2),k(3,1),k(3,3)]
```

Cuando `N` comienza a crecer, puede haber más de una solución, así que su predicado debe estar escrito de manera tal que genere todas las soluciones por *backtracking*.

Nota: El problema de encontrar el número máximo de caballos que caben en un tablero de ajedrez de `N` por `N` sin atacarse está resuelto hace mucho y existe una fórmula cerrada para calcular ese máximo: no puede usarla. Tiene que poner los caballos, hasta encontrar el valor máximo posible.

¿Cómo se ven? (3 puntos)

Escriba el predicado `caballito/2` que será invocado con su primer argumento unificado con el tamaño del tablero, y su segundo argumento unificado con una lista que sigue el formato descriptivo de tableros de ajedrez arriba mencionado.

Si se suministró un número entero positivo *y* la lista que representa el tablero es *válida*, entonces debe triunfar mostrando el tablero en pantalla. Si no se cumplen *ambas* condiciones simultáneamente, debe fallar sin mostrar *nada* en la pantalla.

Esto es, la idea es poder utilizar

```
?- arre(3,N,L), caballito(N,L).
+-+-+-+
|K| |K|
+-+-+-+
| |K| |
+-+-+-+
|K| |K|
+-+-+-+
yes
?- caballito(N,L).
no
?- caballito(2,_).
no
?- caballito(2,[k(1,1),k(1,3)]).
no
```

El primer caso de fallo, se debe que `N` no está instanciada; debería tener el mismo comportamiento si `N` no es un número positivo.

En los dos últimos casos, las listas son obviamente inválidas.

Grafo araña (8 puntos)

Una «araña de cobertura» para un grafo no dirigido cualquiera, es un árbol de cobertura tal que es un «grafo araña». Un grafo es araña si tiene a lo sumo un vértice de grado 3 o más. No todo grafo tiene una araña de cobertura.

Suponga que los siguientes «hechos» describen el grafo `g1` enumerando sus aristas, en cualquier orden

```
edge(g1,a,b).
edge(g1,b,c).
edge(g1,a,c).
edge(g1,d,a).
edge(g1,d,b).
edge(g1,d,c).
```

Escriba el predicado `spider/2` que triunfa si y sólo si, el grafo `G` indicado como primer argumento tiene una araña de cobertura, unificando su segundo argumento con la lista de *aristas* que constituyen tal araña. Esto es,

```
?- spider(g1,L).
L = [edge(a,b),edge(d,a),edge(a,c)]
yes
```

No importa el orden en que salgan las aristas de la araña. Lo que importa es que salgan solamente las aristas de la araña, y que coincidan con las aristas originales (no puede «inventar» aristas).

Un grafo podría tener más de una araña de cobertura, así que su predicado debe estar escrito de manera tal que genere todas las soluciones por *backtracking*.

Mondrian Pizza (9 puntos)

El arte geométrico de Piet Mondrian es particularmente interesante. A partir de 1919 comenzó a producir obras de arte geométrico, basadas en simples rectángulos, algunos rellenos de color, otros no. Ese estilo se convirtió en su firma, que muchos colegas y críticos describen como ideales de paz espiritual. Contemplar una pieza de Mondrian con ojo matemático curioso, debería llevar a preguntarse «¿cómo escogió esos colores?» y «¿por qué dividió el espacio como lo dividió?».

Cambiando de tema, la pizza, sin duda alguna, trae paz espiritual a los que la consumen. Compartir pizza entre amigos siempre es una oportunidad para determinar quién come más o menos, disimulando su apetito detrás de la conversación. Claro que la pizza, en su frecuente presentación circular, no deja espacio para la expresión artística geométrica,

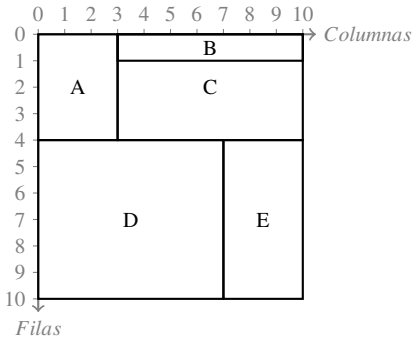
porque generalmente es cortada a través del centro, sobre su diámetro, para producir las convencionales cuñas de círculo. Y cualquier otro corte sería, por decir lo menos, incivilizado.

Hay una historia apócrifa sobre Mondrian trabajando de incógnito en una pizzería de Amsterdam, en la cual se preparaban pizzas cuadradas. Mondrian, siendo Mondrian, cortaba las pizzas como si de sus obras de arte se tratase, para deleite de (algunos) parroquianos. Esto es, siendo la pizza *cuadrada*, Mondrian hacía cortes rectos paralelos a los lados; nunca diagonales.

El dueño de la pizzería, para aumentar el éxito de la propuesta, agregó tres reglas adicionales:

1. Los cortes están separados entre si, o de los bordes, al menos 1 pulgada. Esto, para que cada trozo satisfaga el apetito del consumidor.
2. Todos los trozos tienen que ser rectangulares. En caso de duda (porque me lo han preguntado, y lloré en silencio), un cuadrado es un rectángulo.
3. No puede haber dos trozos de pizza con las mismas *dimensiones*. Esto, porque el dueño de la pizzería es obsesivo compulsivo.

Por ejemplo, una pizza de 10x10 pulgadas podría haber sido cortada



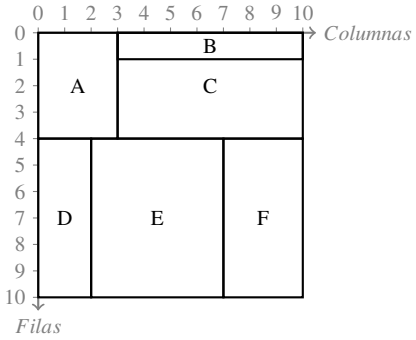
En este corte:

- El segmento `A` es de 4x3.
- El segmento `B` es de 1x7.
- El segmento `C` es de 3x7.
- El segmento `D` es de 6x7.
- El segmento `E` es de 6x3.

Asignaremos una «Puntuación de Mondrian» para calificar cuán bueno es el corte de la pizza, artísticamente hablando. La puntuación es igual al área del trozo más grande,

menos el área del trozo más pequeño. Así, en nuestro primer ejemplo, la puntuación es 42 - 7 = 35.

La misma pizza podría haber sido cortada como

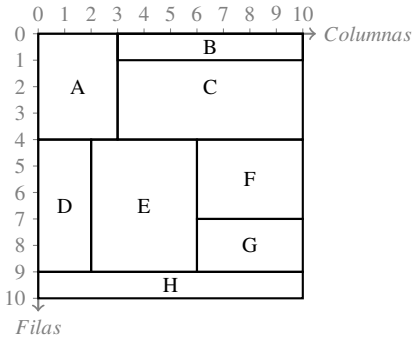


En este corte:

- El segmento `A` es de 4x3.
- El segmento `B` es de 1x7.
- El segmento `C` es de 3x7.
- El segmento `D` es de 6x2.
- El segmento `E` es de 6x5.
- El segmento `F` es de 6x3.

de modo que la «Puntuación de Mondrian» es 30 - 7 = 23, que es mucho mejor que la anterior.

Pero la misma pizza **NO** podría haber sido cortada como



Porque en este corte:

- El segmento `A` es de 4x3.
- El segmento `B` es de 1x7.
- El segmento `C` es de 3x7.
- El segmento `D` es de 5x2.
- El segmento `E` es de 5x4.
- El segmento `F` es de 3x4.
- El segmento `G` es de 2x4.
- El segmento `H` es de 1x10.

Esto rompe con la segunda regla, pues las dimensiones de `A` y `F` son iguales. Note que no hay ningún problema con las **superficies**: la superficie de `D` (5x2) es la misma de `H` (1x10), pero tienen **dimensiones** diferentes que es lo que importa.

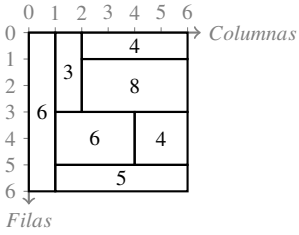
Puede explorar el problema aquí.

Y ésto se relaciona con Prolog… ¿cómo?

Escriba el predicado `mincuts/3` cuyo primer argumento **siempre** estará unificado con el `N` correspondiente a las dimensiones de la pizza, y que debe triunfar unificando su segundo argumento con el **mínimo** puntaje posible para cualquier cortado de la pizza, y su tercer argumento indicando específicamente los trozos a cortar, presentando el resutado como una lista de *slices* (la indentación de la lista es mía, para aclarar la intención)

```
?- mincuts(6,Min,How).
Min = 5
How = [
        slice(0,0,6,1),
        slice(0,1,3,2),
        slice(0,2,1,6),
        slice(1,2,3,6),
        slice(3,1,5,4),
        slice(3,4,5,6),
        slice(5,1,6,6)
      ]
yes
```

Con el functor `slice(F0,C0,F1,C1)` representando el vértice superior izquierdo con fila `F0` y columna `C0`, y el vértice inferior derecho con fila `F1` y columna `C1`, respectivamente. Esta solución, correspondería a los cortes



No importa el orden en que aparezcan los rectángulos de la solución. Así mismo, podría haber más de una solución, pero basta que encuentre **una**. Esto quiere decir que su predicado no tiene que hacer *backtracking* a menos que sea necesario. De hecho, le conviene usar *cuts* para controlar la profundidad de la recursión, o arriesgarse a usar DFS/BFS para hacer la búsqueda (ninguna alternativa es más fácil que la otra, naturalmente).

¿Cómo saber si lo está haciendo bien? El problema ha sido estudiado minuciosamente, y si bien no hay una «fórmula cerrada» para saber calcular el mínimo a partir de `N`, se ha encontrado

N	Mínimo
5	4
6	5
7	5
8	6
9	6
10	8

¡Extra de pepperoni! (5 puntos)

Si Ud. resolvió con éxito este problema, y en efecto encuentra una distribución de cortes con puntaje mínimo, puede optar por puntos adicionales si escribe el predicado `toppings(How,Min)` tal que su primer argumento **siempre** esté unificado con algún corte Mondrian para la pizza, y que triunfe unificando su segundo argumento con la cantidad **mínima** de extras que se pueden poner a la pizza de manera que no haya dos segmentos adyacentes con el mismo **topping**. Se consideran adyacentes dos segmentos que comparten aristas o vértices.

Detalles de la Entrega

La entrega se hará en un archivo `pP-<G>.tar.gz`, donde `<G>` es el número de grupo. El archivo *debe* estar en formato TAR comprimido con GZIP – ignoraré, sin derecho a pataleo, cualquier otro formato que yo puedo descomprimir pero que *no quiero* recibir.

Ese archivo, al expandirlo, debe producir un *directorio* que *sólo* contenga:

- El archivo `mk.pro` con la solución a la sección **Arre caballito**
- El archivo `spider.pro` con la solución a la sección **Grafo Araña**
- El archivo `mondrian.pro` con la solución a la sección **Mondrian Pizza**
- Sus predicados **deben** estar organizados de acuerdo con las mejores prácticas de programación Prolog, y de manera que se vea correctamente en **cualquier** editor.
- Sus archivos **deben** estar completa y correctamente documentados, siguiendo las mejores prácticas de documentación para Prolog, al menos para la descripción de los predicados.
- Su implantación será verificada utilizando GNU Prolog 1.3 sobre Debian GNU/Linux. Si Ud. quiere trabajar con SWI Prolog, puede hacerlo, pero **no** puede emplear predicados específicos provistos por SWI Prolog – asegúrese que sus soluciones funcionan con GNU Prolog antes de entregarlas.

El proyecto debe ser entregado por correo electrónico a mi dirección de contacto a más tardar el miércoles 2016-12-14 a las 23:59. Cada minuto de retraso en la entrega le restará un (1) punto de la calificación final.