Trabajo Práctico 2
Programación Lógica
versión 1.0

Paradigmas ~~de Lenguajes~~ de Programación
2do cuatrimestre 2025

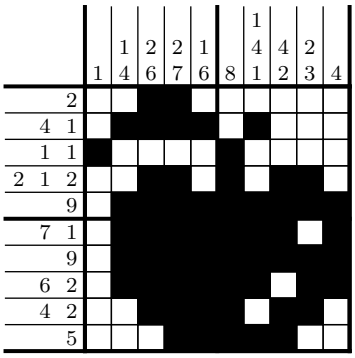
Fecha de entrega: viernes 11 de noviembre

1. Introducción

Los nonogramas son un tipo de rompecabezas lógico que consiste en pintar una matriz mediante ciertas restricciones: los números proporcionados en los bordes. Estos números indican las secuencias de celdas pintadas en cada fila y columna.

Los números en los bordes de las filas y columnas indican la cantidad de celdas consecutivas que deben ser pintadas. Por ejemplo, si una fila tiene los números "3 2", significa que hay un grupo de 3 celdas pintadas seguidas, luego al menos una celda sin pintar, y luego un grupo de 2 celdas pintadas. Antes del grupo de 3 y después del grupo de 2 puede haber cualquier cantidad de celdas sin pintar, incluyendo cero.

A continuación se muestra un ejemplo de nonograma junto con su solución.



Hay algunos nonogramas que se pueden resolver únicamente deduciendo sobre las restricciones, sin necesidad de probar opciones. Depende de qué lógica se use para deducir, cada vez que se pinta una celda, puede ayudar a deducir otras celdas. Las deducciones son determinísticas, no hay elecciones involucradas.

Hay otros nonogramas que requieren probar opciones y eventualmente volver atrás para encontrar la solución correcta. Entre cada paso de probar opciones se puede volver a aplicar la lógica de deducción para pintar más celdas.

Independientemente de cómo se lo resuelva: solamente con deducciones o probando opciones, hay nonogramas que tienen una única solución y otros que tienen múltiples soluciones. Los más satisfactorios son aquellos que tienen una única solución y que además muestran un dibujo claro.

En este link encontrarán muchos nonogramas que pueden tratar de resolver para entender mejor la idea: nonograms.org

1.1. Objetivo

En este trabajo práctico vamos a implementar un programa que resuelva nonogramas en Prolog.

Las estructuras de datos, la lógica que vamos a usar y algunos predicados ya están diseñadas o implementados. Hay que completar los predicados que faltan descritos a continuación.

Finalmente se pide realizar algunas mediciones y analizar la reversibilidad de uno de los predicados implementados.

Se dispone de un `Makefile` que permite cargar el entorno y ejecutar los tests de los ejercicios.

- `make repl` inicia swipl con el archivo `nonograma.pl` cargado.
- `make test` corre los tests de `tests.pl` usando plunit.
- `make watch` corre los tests automáticamente cada vez que se modifica algún archivo `.pl`. Requiere `fswatch` y `xargs`.

1.2. Representación de un nonograma

Un nonograma se representa con el término `nono(M, RS)` donde:

- `M` es una matriz representada como lista de filas (la matriz a pintar).
- Cada elemento de la matriz `M` representa una celda.
- Una celda puede tomar los valores `x` (celda pintada o negra), `o` (celda sin pintar o blanca) o ser una variable no instanciada (color sin determinar aún).
- `RS` es una lista de términos `r(R, L)` que representa las restricciones para las filas y columnas.
- Cada `R` es una lista de enteros.
- Cada `L` es la lista de celdas a las que se aplica la restricción `R`.

Por ejemplo, un nonograma 2 filas y 3 columnas con todas las celdas no instanciadas y las restricciones correspondientes se representa como:

		2	1
1	¿?	¿?	¿?
2	¿?	¿?	¿?

```
nono([[A, B, C],
      [D, E, F]
     ],
     [r([1], [A, B, C]),  % filas
      r([2], [D, E, F]),
      r([], [A, D]),      % columnas
      r([2], [B, E]),
      r([1], [C, F])
     ]).
```

Al ir tratando de resolver el nonograma, las variables no instanciadas se van unificando con `x` o `o` según corresponda y van a quedando instanciadas las celdas tanto en la matriz `M` como en las restricciones `RS`.

Por ejemplo, la primera columna sabemos tienen que ser blancas. Unificando `A=o, D=o` queda:

		2	1
1		¿?	¿?
2		¿?	¿?

```
nono([[o, B, C],
      [o, E, F]
     ],
     [r([1], [o, B, C]),  % filas
      r([2], [o, E, F]),
      r([], [o, o]),      % columnas
      r([2], [B, E]),
      r([1], [C, F])
     ]).
```

De forma análoga, la segunda columna tienen que ser negras. Unificando `B=x, E=x` queda:

		2	1
1			¿?
2			¿?

```
nono([[o, x, C],
      [o, x, F]
     ],
     [r([1], [o, x, C]),  % filas
      r([2], [o, x, F]),
      r([], [o, o]),      % columnas
      r([2], [x, x]),
      r([1], [C, F])
     ]).
```

En los ejemplos anteriores todas las restricciones eran de un solo número. Pero pueden ser más.

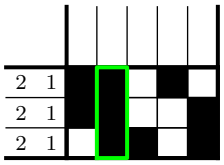
		1 1	2	2 1	1	1
1	1	¿?	¿?	¿?	¿?	¿?
2	1	¿?	¿?	¿?	¿?	¿?
3	1	¿?	¿?	¿?	¿?	¿?
	4	¿?	¿?	¿?	¿?	¿?

```
nono([[A, B, C, D, E],
      [F, G, H, I, J],
      [K, L, M, N, O],
      [P, Q, R, S, T]
     ],
     [r([1,1], [A, B, C, D, E]), % filas
      r([2,1], [F, G, H, I, J]),
      r([3,1], [K, L, M, N, O]),
        r([4], [P, Q, R, S, T]),
      r([1,1], [A, F, K, P]),    % columnas
        r([2], [B, G, L, Q]),
      r([2,1], [C, H, M, R]),
        r([1], [D, I, N, S]),
        r([1], [E, J, O, T])
     ]).
```

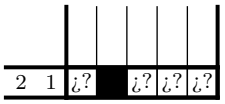
1.3. (Una) lógica para resolver nonogramas

En los ejercicios a continuación vamos a implementar predicados que resuelvan nonogramas a partir de las restricciones de cada fila y columna. A modo de introducción vamos a describir la lógica que vamos a usar.

El siguiente diagrama muestra todas las posibles soluciones para una restricción `[2, 1]` en una fila de longitud 5. Todas las posibilidades tienen en común que la segunda celda debe estar pintada (`x`). Con lo cual podemos pintar esa celda con seguridad, no hay elección.




```
r([2,1], [x,x,o,x,o])
r([2,1], [x,x,o,o,x])
r([2,1], [o,x,x,o,x])
```




```
r([2,1], [A,x,C,D,E])
```

Lo mismo aplica para celdas que necesariamente no deben estar pintadas (`o`) y para las columnas.

La representación de restricciones elegida permite tratar filas y columnas de la misma forma.

Como no todos los nonogramas se pueden resolver sólo con esta lógica, vamos a necesitar usar *backtracking* para probar distintas combinaciones de pintadas válidas. Cuando así lo hagamos, vamos a elegir una de las posibles soluciones para la fila (o columna) que tenga alguna celda no instanciada. Esta decisión puede ser la incorrecta y eventualmente vamos a tener que volver atrás y probar otra opción.

2. Ejercicios

Ejercicio 1: matriz/3

Definir el predicado `matriz(+F, +C, -M)` que es verdadero si `M` es una matriz (lista de listas) de F filas y C columnas. Cuando `M` no está instanciada el predicado debe generar una matriz con variables no instanciadas en las celdas

```
?- matriz(2, 3, M).
M = [[_, _, _], [_, _, _]].
```

Nota: Cada una uno de los _ representa una variable diferente.

Usando `nth1/3` podemos acceder a la primera fila y luego a la segunda celda de esa fila:

```
?- matriz(2, 3, M), nth1(1, M, F1), nth1(2, F1, x).
M = [[_A, x, _B], [_, _, _]],
F1 = [_A, x, _B].
```

Ejercicio 2: replicar/3

Definir el predicado `replicar(+Elem, +N, -Lista)` es cierto cuando `Lista` es una lista de longitud `N`, donde cada elemento es igual a `Elem`.

```
?- replicar(x, 3, L).
L = [x, x, x].
```

Ejercicio 3: transponer/2

Definir el predicado `transponer(+M, -MT)` es cierto cuando `MT` es la matriz transpuesta de `M`. La transpuesta, `MT`, tiene como filas las columnas de `M` y viceversa. Asumir que `M` es una matriz bien formada (todas las filas tienen la misma longitud). `M` puede estar parcialmente instanciada, es decir, puede contener variables no instanciadas como elementos de la matriz.

```
?- transponer([[1, 2, 3], [4, 5, 6]], MT).
MT = [[1, 4], [2, 5], [3, 6]].

?- transponer([[1, 2, A], [4, B, 6]], MT).
MT = [[1, 4], [2, B], [A, 6]].
```

Predicado existente: armarNono/3

`armarNono(+RF, +RC, -NN)` genera un nonograma a partir de las restricciones de filas `RF` y las restricciones de columnas `RC` con todas las celdas no instanciadas.

`RF` y `RC` son listas de listas de enteros, donde cada lista de enteros representa las restricciones de una fila o columna respectivamente.

```
?- armarNono([[1],[2]],[[],[2],[1]], NN).
NN = nono([
           [_A, _B, _C],
           [_D, _E, _F]
          ],
          [
            r([1], [_A, _B, _C]),
            r([2], [_D, _E, _F]),
            r([], [_A, _D]),
            r([2], [_B, _E]),
            r([1], [_C, _F])
          ]).
```

Nota: Éste predicado usa `matriz/3` y `transponer/2`. Si no funciona de la manera esperada, asegúrense de que esos predicados estén correctamente implementados.

Predicado existente: nn/2

Se dispone de varios nonogramas predefinidos con el predicado `nn/2`. `nn(+N, -NN)` genera el nonograma número `N` en la variable `NN`. Utiliza `armarNono/3` internamente. Pueden agregar sus propios nonogramas siguiendo el mismo formato.

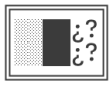
No se pueden modificar los nonogramas predefinidos.

```
?- nn(0, NN).
NN = nono([[_A, _B, _C], [_D, _E, _F]], [r([1], [_A, _B, _C]) | ...]) .
```

Predicado existente: mostrarNono/1, mostrarFila/1

Se dispone de `mostrarNono/1` que muestra un nonograma en la consola. En el siguiente ejemplo definimos manualmente el valor de las celdas del nonograma usando la información de las restricciones de las primeras dos columnas. Es un ejemplo a modo ilustrativo, para ver cómo se muestra un nonograma.

```
?- nn(0, NN), NN=nono([[o,x,_],[o,x,_]], _), mostrarNono(NN).
```



```
NN = nono([[o, x, _A], [o, x, _B]], [r([1], [o, x, _A]), r([2], [o, x, _B]), r([], [o, o]),
r([2], [x, x]), r([1], [_A, _B])]) .
```

También se puede usar `mostrarFila/1` para mostrar una fila individualmente.

Ejercicio 4: pintadasValidas/1

Definir el predicado `pintadasValidas(+R)` que genera las posibles pintadas válidas para una restricción `R`. `R` será de la forma `r(Res, Celdas)` donde `Res` es una lista de enteros que representan las restricciones y `Celdas` es una lista de variables (parcialmente no instanciadas) que representan las celdas. Notar que `R` puede corresponder tanto a una fila como a una columna. La estructura del nonograma nos permite tratarlas de la misma forma.

Importante: No debe generar soluciones repetidas.

```
?- length(L, 5), pintadasValidas(r([3], L)), mostrarFila(L).
L = [x, x, x, o, o] ;
L = [o, x, x, x, o] ;
L = [o, o, x, x, x] ;
false.
```

Debe poder funcionar con celdas parcialmente instanciadas en `x` o `o`.

```
?- L=[_,_,x,_,_], pintadasValidas(r([1,1], L)), mostrarFila(L).
```

```
L = [x, o, x, o, o] ;
```

```
L = [o, o, x, o, x] ;
false.
```

También debe poder funcionar para verificar si hay alguna pintada válida dada.

```
?- L=[_,_,x,x,_], pintadasValidas(r([1,1], L)), mostrarFila(L).
false.
```

Tip 1: Se puede usar el predicado `replicar/3` definido anteriormente.

Tip 2: Dada una lista de restricciones `[R|RS]` y una lista de celdas `L` de longitud `N` sabemos que antes de las `R` celdas pintadas puede o no haber celdas sin pintar. Entre las demás restricciones debe haber al menos una celda sin pintar. Esto limita cuántas celdas sin pintar puede haber en total. Puede resultar conveniente plantear un predicado auxiliar que resuelva en forma recursiva y que reciba como parámetro adicional la cantidad mínima de celdas sin pintar que debe haber antes de la primera restricción. También pueden considerar pasar como parámetro la sumatoria de las restricciones para evitar calcularla varias veces.

Tip 3: Otra alternativa es pensar el problema como un problema de combinatoria. Sabemos exactamente cuántas celdas deben ser pintadas y cuántas no pintadas. Dada una lista de restricciones $r_1, r_2, \ldots, r_k$ aplicada a una lista de tamaño N sabemos que la cantidad total de celdas pintadas debe ser $\sum_{i=1}^{k} r_i$ y la cantidad total de celdas no pintadas debe ser $N - \sum_{i=1}^{k} r_i$. También sabemos que entre cada par de restricciones debe haber al menos una celda no pintada, pero que antes y después puede haber cero o más celdas no pintadas. Lo cual significa que tenemos que elegir $b_0, b_1, \ldots, b_k$ que serán las cantidades de celdas no pintadas. $\sum_{i=0}^{k} b_i = N - \sum_{i=1}^{k} r_i$, $b_1, b_2, \ldots, b_k \geq 1$, $b_0, b_k \geq 0$. Una vez que tenemos estos valores podemos armar la lista final $[b_0, r_1, b_1, r_2, b_2, \ldots, r_k, b_k]$.

Ejercicio 5: resolverNaive/1

Definir el predicado `resolverNaive(+NN)` que resuelve un nonograma `NN` usando *backtracking* y el predicado `pintadasValidas/1` definido anteriormente.

```
?- nn(0, NN), resolverNaive(NN), mostrarNono(NN).
```

```
NN = nono([[o, x, o], [o, x, x]], [r([1], [o, x, o]), r([2], [o, x, x]), r([], [o, o]),
r([2], [x, x]), r([1], [o, x])]) ;
false.
```

Nota: Puede que para tamaños medios sea muy lenta.

Ejercicio 6: pintarObligatorias/1

Definir el predicado `pintarObligatorias(+R)` que pinta las celdas que son obligatoriamente `x` o `o`. Esto es viendo todas las posibilidades de pintadas válidas para la restricción `R`, tal como se ilustraba en la sección 1.3. `R` será de la forma `r(Res, Celdas)` donde `Res` es una lista de enteros que representan las restricciones y `Celdas` es una lista de variables (parcialmente no instanciadas) que representan las celdas.

```
?- length(L, 5), pintarObligatorias(r([5], L)), mostrarFila(L).
```

```
L = [x, x, x, x, x] ;
false.

?- length(L, 5), pintarObligatorias(r([], L)), mostrarFila(L).
```

```
L = [o, o, o, o, o] ;
false.

?- length(L, 5), pintarObligatorias(r([3], L)), mostrarFila(L).
```

```
L = [_, _, x, _, _] ;
false.

?- length(L, 5), pintarObligatorias(r([4], L)), mostrarFila(L).
```

```
L = [_, x, x, x, _] ;
false.
```

También debe poder funcionar para verificar si hay alguna pintada válida dada.

```
?- L = [_,o,_,_,_], pintarObligatorias(r([4], L)), mostrarFila(L).
false.
```

Tip: Traten de combinar todas las posibles pintadas válidas para encontrar las celdas obligatorias.

Retomando el ejemplo inicial de restricción `[2, 1]` para una lista de 5 celdas sabemos que las posibilidades son:

- `[x, x, o, x, o]`
- `[x, x, o, o, x]`
- `[o, x, x, o, x]`

Combinando las primeras dos posibilidades tenemos `[x, x, o, D, E]`, y combinando este resultado con la tercera posibilidad tenemos `[A, x, C, D, E]`.

Se dispone del predicado `combinarCelda/3`. `combinarCelda(?A, ?B, ?C)` es verdadero cuando el valor de `A` y `B` es consistente con `C`.

```
?- combinarCelda(x, x, C).
C = x ;
false.

?- combinarCelda(o, o, C).
C = o ;
false.

?- combinarCelda(x, o, C).
true.

?- combinarCelda(_, o, C).
true ;
false.
```

Ejercicio 7: deducir1Pasada/1

Definir el predicado `deducir1Pasada(+NN)` que aplica el predicado `pintarObligatorias/1` a todas las restricciones del nonograma `NN`.

Es posible que este predicado logre resolver varios nonogramas sencillos por sí solo, pero no todos.

Predicado existente: cantidadVariablesLibres/2

Se dispone del predicado `cantidadVariablesLibres(?T, -N)` que es verdadero cuando `N` es la cantidad de variables no instanciadas en el término `T`.

Puede ser usado para contar la cantidad de celdas no instanciadas en una lista, en una matriz, en un nonograma o en cualquier término.

```
?- cantidadVariablesLibres([x,o,C,D,E],N).
N = 3.
```

Predicado existente: deducirVariasPasadas/1

Se dispone del predicado `deducirVariasPasadas(+NN)` que aplica el predicado `deducir1Pasada/1` repetidamente hasta que no pueda deducir más celdas.

En la implementación se puede ver que se usa `cantidadVariablesLibres/2` para contar las celdas no instanciadas antes y después de aplicar `deducir1Pasada/1`.

Luego, `deducirVariasPasadasCont/3` es un predicado auxiliar que recibe estas cantidades y dependendiendo de si coinciden o no itera o frena.

Definir un predicado que define si continuar o no puede ser útil en otro ejercicio `;-)`.

Ejercicio 8: restriccionConMenosLibres/2

Definir el predicado `restriccionConMenosLibres(+NN, -R)` que es verdadero cuando `R` es la restricción (o una de las restricciones) del nonograma `NN` que tiene la menor cantidad de celdas no instanciadas, pero que tenga al menos una celda no instanciada. Este predicado va a ser usado para elegir la siguiente restricción a resolver cuando se use backtracking, por eso es importante que tenga al menos una celda libre.

Importante: El predicado no puede ser recursivo. Aprovechen el `not/1`. **No se permite el uso de `findall/3`, `bagof/3`, `setof/3` ni ningún otro metapredicado salvo `not/1`.**

Ejercicio 9: resolverDeduciendo/2

Definir el predicado `resolverDeduciendo(+NN)` que resuelve un nonograma `NN` de manera más eficiente que `resolverNaive/1`. Para ello debe comenzar por resolver usando `deducirVariasPasadas/1`, si no está resuelto, elegir una restricción con menor cantidad de celdas libres usando `restriccionConMenosLibres/2` y probar las posibles pintadas válidas para esa restricción usando `pintadasValidas/1`. Continuar aplicando la misma lógica, una vez que se elige una pintada válida, se puede volver a usar `deducirVariasPasadas/1` para aprovechar las nuevas celdas pintadas.

Notar que inmediatamente después de aplicar `pintadasValidas/1` no se garantiza que la solución parcial sea consistente con las demás restricciones.

Importante: Este predicado no debe generar soluciones repetidas, para ello se permite el uso de `!/1` (cut).

Ejercicio 10: solucionUnica/1

Definir el predicado `solucionUnica(+NN)` que es verdadero cuando el nonograma `NN` tiene una única solución.

No es necesario que el predicado encuentre la solución, alcanza con que determine si es única o no.

Ejercicio 11: Análisis de nonogramas

Se pide completar la siguiente tabla con el análisis de los nonogramas predefinidos. Completar dicha tabla en un comentario en el código fuente o un `.txt`/ texttt.md/`.pdf` que lo acompañe.

Indicar qué consultas se usaron para averiguar cada uno de los datos. Ejemplo: el predicado `tam/2` puede ser usado para completar la primera columna.

```
tam(N, (F, C)) :- nn(N, nono(M, _)), matriz(F, C, M).

?- tam(0, T).
T = (2, 3).
```

Si algún nonograma no puede ser analizado, indicar el motivo.

N	Tamaño	¿Tiene solución única?	¿Es deducible sin backtracking?
0	2×3	Sí	Sí
1	...	...	...
2	...	...	...
3	...	...	...
4	...	...	...
...	...	...	...

Ejercicio 12: Reversibilidad

Indicar si el predicado `replicar/3` es reversible en el segundo argumento. En concreto se pide analizar si `replicar(+Elem, -N, -Lista)` funciona correctamente.

3. Pautas de Entrega

Importante: Se espera que la elaboración de este trabajo sea 100 % de los estudiantes del grupo que realiza la entrega. Así que, más allá de que pueden tomar información de lo visto en las clases o consultar información en la documentación de SWI-Prolog u otra disponible en Internet, no se podrán utilizar herramientas para generar parcial o totalmente en forma automática la resolución del TP (e.g., ChatGPT, Copilot, etc). En caso de detectarse esto, el trabajo será considerado como un plagio, por lo que será gestionado de la misma forma que se resuelven las copias en los parciales u otras instancias de evaluación.

Se deberá subir el código fuente a la tarea respectiva en el Campus.

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 que correspondan.
- No está permitido usar macros que colapsen cláusulas, como `;` y `->`.
- **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.

4. Referencias y sugerencias

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

Se recomienda que, siempre que sea posible, utilicen los predicados ISO y los de SWI-Prolog ya disponibles. Recomendamos especialmente examinar los predicados y metapredicados que figuran en la sección *Útil* 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.

En particular, seguramente les resulten útiles los siguientes predicados: `between/3`, `member/2`, `length/2`, `append/3`, `maplist/2`, `maplist/4`, `findall/3`, `not/1`.