Lenguajes Declarativos

Informe - TPO 1

Alumnos:
Sebastián Alejandro Reibold
Bautista Fernandez Gramajo
Nicanor Fernandez

Profesores:
Claudio Vaucheret
Giuliano Marinelli

Índice

1. Introduccion

El presente informe explica la resolución de los problemas planteados en el Trabajo Práctico Obligatorio N° 1 de la asignatura Lenguajes Declarativos (2025) de la Facultad de Informática de la Universidad Nacional del Comahue. El objetivo principal de este trabajo es aplicar los conceptos de programación lógica con restricciones, utilizando el lenguaje Prolog, para encontrar soluciones a diversos problemas.

Se abordan cuatro ejercicios distintos: un problema de posicionamiento estratégico en ajedrez, un acertijo lógico deductivo sobre personas, oficios y ciudades, un criptoaritmo y un problema de planificación con restricciones de tiempo y capacidad (el cruce del puente). Para cada uno, se presentarán las soluciones obtenidas mediante la ejecución del programa correspondiente (adjuntados dentro de la carpeta comprimida entregada).

Para este trabajo se utilizó **SWI-Prolog**, versión 9.2.9.

2. Ejercicio N° 1

Nuestra solución propone un algoritmo que comienza representando las posiciones de las piezas de ajedrez (reina, alfil, caballo y torre) mediante variables que indican sus filas y columnas dentro del tablero 8x8. Sin embargo, para facilitar la verificación de que no ocupen la misma casilla, estas posiciones se 'linealizan' a valores of 1 to 64, y con esto garantizamos, de una manera sencilla, que ninguna pieza se interponga con otra usando **'all_different'**.

Luego, se define una lista de casillas marcadas, cada una con su ubicación (fila y columna) y la cantidad exacta de piezas que deben atacarla. Para el caso del ejercicio extra, esta es la única sección de código la cual debimos modificar acorde al enunciado.

Luego, se aplica un módulo auxiliar que evalúa, para cada casilla marcada, si es atacada por cada una de las piezas. Se utilizan las reglas clásicas del ajedrez para definir el ataque: la reina ataca si comparte fila, columna o diagonal, el alfil si está en una diagonal, el caballo si está a una distancia tipo 'L' (2x1 o 1x2) y la torre si comparte fila o columna. Cada posible ataque se representa con una variable booleana.

Finalmente, se suma el total de ataques por casilla y se fuerza a que coincida con el valor indicado. Una vez aplicadas todas las restricciones, se realiza "labeling" para encontrar los valores concretos que cumplen con todas las condiciones. Luego, se devuelven las posiciones de las piezas de forma legible.

3. Ejercicio N° 2

El ejercicio 2 nos brinda 5 nombres de hombres, 5 ciudades en las que pueden residir y 5 profesiones que pueden poseer y nos pregunta si podríamos determinar la ciudad en la que reside uno de estos hombres (Nash) basándonos en ciertas restricciones especificadas en el ejercicio.

Como primera instancia para la realización del ejercicio buscamos determinar en qué ciudad reside y qué profesión posee cada uno de los hombres, para esto buscamos relacionar cada uno de los nombres, ciudades y profesiones con un número del 1 al 5. Una vez hecho esto podríamos determinar la ciudad y profesión de cada hombre verificando la igualdad de su número.

Para realizar esta relación comenzando representando los nombres, profesiones y ciudades en un formato "Variable-instancia", para así poder instanciar la variable en un número entero aplicando sobre esta las restricciones correspondientes. Para facilitar este mapeo utilizamos la librería **"pairs"**. Adjunto a continuación ejemplo de este mapeo para el conjunto de los nombres.

```
Nombres = [Green, Brown, Peters, Harper, Nash],
NombresN = [green, brown, peters, harper, nash],
pairs_keys_values(ParesNombres, Nombres, NombresN),
```

Una vez creada la lista de pares, comenzamos a aplicar las restricciones necesarias sobre las variables, comenzando por definir que no pueden repetirse números en el mismo conjunto, es decir, no puede haber dos nombres con el mismo número. Realizamos esta restricción con el predicado **"all_distinct"**.

Luego de esto concatenamos las listas que contienen las variables utilizadas para los pares en una lista llamada Resultado y definimos un dominio de 1 a 5 sobre esta variable, esto nos indicaría que cada elemento de Resultado va a estar instanciado por un número del 1 al 5.

Una vez conseguida la lista de las variables, comenzamos a aplicar las restricciones que propone el ejercicio mediante operadores de igualdad y desigualdad de enteros.

Una vez aplicadas todas las restricciones, la variable Pares resulta en una lista de listas, donde cada una contiene los pares Número-nombre, Número-ciudad y Número-profesión, respectivamente.

Una vez realizado esto, se busca una solución concreta del problema con el predicado **"label"**. Luego, de la lista pares obtenemos la lista de nombres y de ciudades utilizando el predicado **"nth1"** en la posición 1 y 2 respectivamente. Una vez obtenidas las listas utilizo el predicado **"member"** para buscar el número que corresponde a nash en

la lista de nombres, una vez instanciado el número utilizo el mismo predicado para instanciar la ciudad correspondiente a ese número en la lista de ciudades, dando como instancia de la variable Ciudad, a la ciudad que corresponde a Nash. Una vez realizado esto, imprimimos esa ciudad por pantalla, finalizando el problema.

4. Ejercicio N° 3

Nuestra solución se enfoca en verificar la validez de una suma criptográfica donde cada letra representa un dígito distinto del 0 al 9. El problema plantea la siguiente suma:

```
FELIZ + DIA + DEL = PADRE
```

Para resolver este problema utilizando la librería `clpfd`, se implementan tres restricciones principales:

1. **Restricción de unicidad**: Cada letra representa un dígito distinto. Esto se implementa con el predicado:

   ```
   all_distinct([F, E, L, I, Z, D, A, P, R]).
   ```

2. **Restricción de suma numérica**: Cada palabra se convierte a su valor posicional:

   $$\begin{aligned} \texttt{FELIZ} &= F \times 10000 + E \times 1000 + L \times 100 + I \times 10 + Z \\ \texttt{DIA} &= D \times 100 + I \times 10 + A \\ \texttt{DEL} &= D \times 100 + E \times 10 + L \\ \texttt{PADRE} &= P \times 10000 + A \times 1000 + D \times 100 + R \times 10 + E \end{aligned}$$

   La restricción se define con el operador `#=`:

   ```
   FELIZ + DIA + DEL #= PADRE.
   ```

3. **Restricción de no ceros iniciales**:

   ```
   F #\= 0, D #\= 0, P #\= 0.
   ```

Una vez definidas las restricciones, se utiliza el predicado `label/1` para asignar valores:

```
label([F, E, L, I, Z, D, A, P, R]).
```

Este proceso aplica *backtracking* automático para encontrar combinaciones válidas, aprovechando la potencia declarativa de Prolog y la eficiencia de `clpfd` para manejar restricciones aritméticas y de dominio.

5. Ejercicio N° 4

Este ejercicio plantea una problemática donde, dado el tiempo que tarda cada personaje en cruzar un puente, sabiendo que únicamente pueden cruzar dos al mismo tiempo y que uno de ellos lleva una linterna necesaria para cruzar, por ende luego de cada cruce uno de los personajes debería volver para que otros puedan cruzar, debemos proponer un orden de cruces para que todos puedan cruzar antes de un límite de tiempo, en este caso 60 minutos.

Para implementar la solución a este problema comenzamos definiendo los tiempos de cada personaje con el predicado **"tiempo"**, luego implementamos los predicados **"cruzar_ida"**, y **"cruzar_vuelta"**, cada uno de estos posee como argumento un predicado **"state"**, este predicado mantiene 3 estructuras, la primera de estas es la lista de personajes que está en la costa 1, es decir, que todavía no cruzó el puente, la segunda es la lista de personajes que ya cruzó el puente, y la tercera es el tiempo que lleva transcurrido desde que comenzaron los cruces.

Los predicados para cruzar de ida y de vuelta se implementan exactamente de la misma manera, con la única diferencia de que el que se utiliza para cruzar de ida admite que crucen dos personajes, en cambio para cruzar de vuelta esto no tendría sentido en el enunciado, por ende únicamente permite un personaje. A su vez, estos están implementados en forma de DCG ya que esto facilita mantener la lista con el resultado.

El predicado **"cruzar_ida"** comienza seleccionando dos personajes de la lista Costa1, utilizando el predicado **"select"**, entre estos dos personajes realiza la operación `\@<"`, lo que evita duplicados en las parejas. Seguido de esto se verifican los tiempos, asumimos que el tiempo que tardan en cruzar el puente lo define el más lento de los dos personajes, por esto hacemos una suma del tiempo acumulado con el máximo de los tiempos de los personajes.

Por último, agregamos a la lista que representa nuestro resultado una cadena de caracteres que dice los personajes que cruzan de ida y luego se realiza un llamado al predicado **"cruzar_vuelta"**, con las estructuras resultantes del cruce de ida.

En el predicado cruzar vuelta se implementa el caso base, el cual se da cuando la lista correspondiente a la costa 1 está vacía, es decir, cuando todos los personajes cruzaron el puente. Como fue dicho anteriormente, este predicado se implementa de la misma manera que el anterior con la única diferencia de que se selecciona únicamente un personaje y realiza el llamado sobre el predicado de **"cruzar_ida"**, operando de una forma recursiva pero cruzada entre estos dos predicados.

Además de todo esto , realizamos un predicado llamado **"ejercicio4"**, en el cual implementamos el primer llamado con el predicado **"phrase"** para que se acepte la DCG y con las estructuras dentro del state en su estado inicial. Llamando a este predicado

nos daría el orden de cruces en la variable Resultado.

6. Conclusión

A lo largo de este trabajo práctico, se ha demostrado la aplicabilidad y eficacia de la programación lógica con restricciones en Prolog para resolver una variedad de problemas. Mediante la definición declarativa de variables, dominios y restricciones, fue posible modelar y encontrar soluciones para los cuatro ejercicios propuestos.

Se cumplieron los objetivos del Trabajo Práctico Obligatorio N° 1, logrando implementar soluciones funcionales para cada uno de los ejercicios planteados.