TRABAJO FIN DE GRADO


FACULTAD DE MATEMÁTICAS

Departamento de Ciencias de la Computación e Inteligencia Artificial

---

# Una introducción a la programación con conjuntos de respuesta. Aplicaciones

---

*Presentado por*:
Marina Jiménez Núñez

*Dirigido por*:
Andrés Cordón Franco

María José Hidalgo Doblado

# Índice general

# Abstract

The main objective of this paper is to present the answer set programming paradigm as a tool to model and solve combinatorial problems, in general, and especially NP-complete or NP-hard problems. First, in chapters 1 and 2, we are going to focus on carrying out a theorical development to help us learning how to program in ASP and, in particular, in CLINGO, and some properties that ASP programs have. In the following chapters, 3 and 4, we are going to study representations of real situations and problems to apply these results then to relevant mathematical problems in chapter 5. Finally, in chapter 6, we study puzzles in which deciding whether there is a solution to them is a NP- complete problem.

# Agradecimientos

*En primer lugar, me gustaría agradecer a mis tutores Andrés Cordón Franco y María José Hidalgo Doblado la ayuda y el apoyo que me han ofrecido durante todo el proceso de realización de este trabajo.*

*En segundo lugar, quisiera agradecer a mis compañeros la infinita paciencia que han demostrado tener y el haberme ayudado siempre. Gracias a los que han empezado esta etapa conmigo y a los que he encontrado por el camino y me han acompañado hasta el fin de la misma.*

*Por último, dar las gracias a mis hermanos y a mis padres. Nunca encontraré las palabras para agradeceros el cariño que he recibido durante todos estos años. Gracias por confiar en mí hasta cuando yo misma no lo hice. Sois mi mayor suerte, mi mayor orgullo y mi ejemplo a seguir por y para siempre.*

# Introducción

El principal objetivo del presente trabajo es presentar el paradigma de la programación con conjuntos de respuestas como una herramienta para modelizar y resolver problemas combinatorios, en general, y muy especialmente problemas NP-completos o NP-duros. En la actualidad, es generalmente aceptado que la clase de complejidad computacional P, que comprende los problemas computacionales resolubles por algún algoritmo en tiempo polinomial, captura la noción de "problema tratable" (esto es, problemas que pueden resolverse en un tiempo de ejecución "razonable" incluso para entradas de tamaño grande). La clase de complejidad NP comprende aquellos problemas computacionales tales que, dado un candidato, *verificar* si es o no una solución del problema puede hacerse en tiempo polinomial (pero, a priori, no se sabe si *encontrar* una solución del problema puede también conseguirse en tiempo polinomial). Determinar si las clases P y NP son o no iguales es el famoso problema P versus NP, uno de los principales problemas abiertos en la Matemática actual. Dentro de la clase NP, destacan un tipo de problemas muy relevantes: los problemas NP-completos. Un problema A se dirá NP-completo si: 1) pertenece a la clase NP y 2) cualquier otro problema en la clase NP es reducible (en tiempo polinomial) a dicho problema A. Los problemas NP-completos constituyen, por tanto, la clase de los problemas "más difíciles" dentro de la clase NP. Es bien conocido que hay multitud de problemas matemáticos interesantes que resultan ser NP-completos. Más generalmente, un problema A se dirá NP-duro (o NP-hard) si cualquier problema en la clase NP es reducible (en tiempo polinomial) a dicho problema A, aunque ahora A no tiene por qué pertenecer a NP. De nuevo, la clase de problemas NP-duros captura la idea de "problemas computacionales presumiblemente difíciles de resolver". La programación con conjuntos de respuesta (o Answer Set Programming, cuyo acrónimo es ASP) es una programación distinta a la tradicional (se enmarca en el campo de la programación declarativa) cuyo principal enfoque es buscar solución a problemas de búsqueda difíciles, principalmente problemas NP-completos o NP-duros.

Para el desarrollo del trabajo, necesitamos adentrarnos un poco más en la programación con conjuntos de respuesta, es decir, necesitamos saber qué son los programas de ASP, qué elementos los forman y qué propiedades tienen estos programas y sus soluciones, los denominados conjuntos de respuesta. Esto es lo que se estudia en los capítulos 1 y 2 de este trabajo. En el capítulo 1, abordamos la sintaxis y la semántica, tanto formal como informal, de los programas de ASP, en la que se exponen algunos ejemplos para ilustrar el contenido. Además, en el capítulo se habla también acerca de la sintaxis de los programas de ASP en CLINGO, sistema que usaremos a lo largo del trabajo para representar y resolver los problemas que se van contemplando. En el capítulo 2, nos centramos en estudiar las propiedades de los programas de ASP y de sus soluciones. Para el capítulo 1 y 2, se usa principalmente información procedente de [GK14], [Lif08], [Lif19], [GKK$^+$19], [GL88] y los apuntes de la página web [HD21] que mencionamos en la bibliografía.

Para abordar nuestro objetivo, la búsqueda de soluciones para problemas NP-hard, necesitamos aprender a representar problemas reales. Nos centramos en esto mismo en el capítulo 3: en este, vemos cómo construir las denominadas bases de conocimiento, en las que se estudia cómo representar situaciones reales, a través de diversos ejemplos, en los que intentamos mostrar la importancia de la representación de todo el conocimiento, incluido el que se posee por sentido común, y también se muestra cómo representar situaciones en las que la información no es completa o posee una estructura jerárquica. En el capítulo 4, se estudian las situaciones en las que aparecen defectos, es decir, situaciones que se rigen por expresiones como generalmente, habitualmente, etc, y las excepciones que nos podemos encontrar en este tipo de circunstancias, recuperando los ejemplos que se exponen en el capítulo 3 y añadiéndoles algunos defectos. Para estas secciones, seguimos principalmente el libro [GK14] y la página web [HD21].

Mientras que en los capítulos 3 y 4 nos centramos principalmente en la representación del problema o situación que se contempla y en encontrar elementos que verifiquen ciertas reglas, en el capítulo 5, el enfoque cambia: el objetivo es buscar soluciones a problemas a través de reducirlos a encontrar programas de ASP cuyos conjuntos de respuesta nos den estas mismas soluciones. Exponemos cómo se pueden plantear estos programas con el estudio de un problema matemático famoso, el encontrar los ciclos hamiltonianos de un grafo, y un puzzle, el sudoku. Para este capítulo, se vuelve a usar tanto la página web [HD21] como el libro [GK14].

Finalmente, en el capítulo 6, planteamos algunos puzzles interesantes en

la programación con conjuntos de respuesta desde un punto de vista computacional y dada su representación. Estos puzzles son el Nurikabe, el puzzle Heyawake y por último, el puzzle Masyu. Con estos puzzles, tratamos totalmente nuestro objetivo, ya que decidir cuándo hay una solución para estos puzzles y cuándo no constituye un problema NP-completo (problema NP y NP-hard al mismo tiempo). Para este capítulo, se sigue muy de cerca la referencia [CKK+07]. Para ilustrar nuestros programas, se han resuelto algunos ejemplos de puzzles extraídos de aplicaciones y páginas web para la generación aleatoria de los mismos.

Para este trabajo, no se necesitan conocimientos previos sobre la programación con conjuntos de respuesta ni acerca del sistema CLINGO, pues el desarrollo teórico necesario para la comprensión del trabajo se realiza a lo largo del mismo.

# Capítulo 1

# Introducción a CLINGO

CLINGO es un sistema desarrollado en la Universidad de Postdam con la finalidad de representar y resolver problemas en el marco de la programación de conjunto de respuesta (en inglés, Answer Set Programming, de donde proceden las siglas ASP). En este capítulo, nos centraremos en presentar los elementos esenciales de este lenguaje para poder entender los restantes capítulos de este trabajo. Para una información completa de este sistema se puede visitar la página web: `https://potassco.org`.

## 1.1. Sintaxis y semántica de programas ASP

ASP es un tipo de programación declarativa orientada a la resolución de problemas combinatorios. A través de un conjunto de reglas, se describen los objetos de un dominio y las relaciones que hay entre estos objetos.

Para poder definir qué es un programa de ASP, veamos primero los elementos que pueden aparecer en estos programas, que son las *constantes*, las *funciones*, los *predicados* y las *variables*. El conjunto de los nombres de las constantes del programa se denominará $O$, y de igual forma, denotaremos como $F$, $P$ y $V$ al conjunto de los nombres de las funciones, los predicados y las variables respectivamente. Los predicados expresan las relaciones que hay entre los objetos o propiedades de los mismos. Se denomina *signatura* a $\Sigma =< O, F, P, V >$

Veamos algunos ejemplos.

**Ejemplo 1.**

Sea el programa P:

```
bloque(a).
bloque(b).
bloque(c).
sobre(a,b).
sobre(b,c).
encima(X,Y) ← sobre(X,Y).
encima(X,Y) ← sobre(X,Z),encima(Z,Y).
```

Tenemos entonces que la signatura $\Sigma_P =< O, F, P, V >$ donde:

- $O = \{a, b, c\}$.
- $F = \emptyset$.
- $P = \{\mathbf{bloque/1}, \mathbf{sobre/2}, \mathbf{encima/2}\}$.
- $V = \{X, Y, Z\}$.

Las variables aparecen en el programa en mayúsculas, mientras que las constantes aparecen en minúsculas. Los predicados y las funciones vienen acompañados de su aridad (número de argumentos en los que se evalúa la función o predicado). La aridad se representa con "*nombre_del_predicado/aridad.*"

**Ejemplo 2.**

Consideremos ahora el programa Q:

```
nat(0).
nat(s(X)) ← nat(X).
```

En este caso, tenemos que $\Sigma_Q$ está formado por:

- $O = \{0\}$.
- $F = \{\mathbf{s/1}\}$.
- $P = \{\mathbf{nat/1}\}$.
- $V = \{X\}$.

Sigamos estudiando los elementos de los programas de ASP.

A partir de los elementos que se han expuesto, se definen los *términos*. Un término se define de manera recursiva de la siguiente forma:

- Cualquier variable o constante es un término.
- Sean $t_1, \ldots, t_n$ términos y sea $f$ una función que pertenece a $F$ y tiene aridad n, entonces $f(t_1, \ldots, t_n)$ es también un término.

Denominamos términos *básicos* a los términos que carecen de variables.

**Continuación de los ejemplos 1 y 2.**

- Los términos del programa P son $\{a, b, c, X, Y, Z\}$, pues como hemos dicho, tanto las constantes como las variables son términos. Entre ellos, los términos $\{a, b, c\}$ son términos básicos del programa, pues carecen de variables.
- El programa Q contiene a la función **s/1**, luego los términos del programa en este caso constituyen un conjunto infinito:

  $$\{0, X, s(0), s(X), s(s(0)), s(s(X)), \ldots\}.$$

  El conjunto de términos básicos es también un conjunto infinito:

  $$\{0, s(0), s(s(0)), \ldots\}.$$

A partir de los términos y los predicados definimos los *átomos*. Sean $t_1, \ldots, t_n$ n-términos y $p$ un símbolo de predicado, entonces $p(t_1, \ldots, t_n)$ es un átomo. Un *literal* es un átomo $p(t_1, ...t_n)$ o su negación $\neg p(t_1, ...t_n)$ (esto es, el complementario del átomo). Para referirnos a la negación del átomo, hablaremos de literal *negativo* y en otro caso, *positivo*. Como ya dijimos, los términos pueden no contener variables. Si los términos que aparecen en el átomo $p(t_1, ...t_n)$ son términos básicos, entonces diremos que el átomo es básico. Así, si consideramos los átomos básicos y sus negaciones obtenemos lo que denominamos literales básicos.

**Continuación de los ejemplos 1 y 2.**

En el caso del programa P, los átomos constituyen el conjunto:

$\{bloque(X), bloque(Y), bloque(Z), bloque(a), bloque(b), bloque(c), sobre(X,X),$
$sobre(X,Y), \ldots, sobre(Z,Z), sobre(a,a), sobre(a,b), \ldots, sobre(c,c),$
$encima(X,X), encima(X,Y), \ldots, encima(a,a), encima(a,b), \ldots, encima(c,c)\}$.

Los átomos básicos del programa son:

*{bloque(a),bloque(b),bloque(c), sobre(a,a),sobre(a,b),. . . , sobre(c,c),*
*encima(a,a),encima(a,b),. . . ,encima(c,c)}.*

Los literales del programa son los átomos y sus complementarios:

*{bloque(X),¬bloque(X),. . . , bloque(a),¬bloque(a),. . . , sobre(X,X),*
*¬sobre(X,X),. . . ,sobre(a,a),¬sobre(a,a),. . . ,encima(X,X),*
*¬encima(X,X),. . . , encima(a,a),¬encima(a,a),. . . ,¬encima(c,c)}.*

Por último, los literales básicos son los átomos básicos y sus complementarios:

*{bloque(a),¬bloque(a),. . . ,sobre(a,a),¬sobre(a,a),. . . ,encima(a,a),*
*¬encima(a,a),. . . ,encima(c,c), ¬encima(c,c)}.*

De manera análoga, podemos ver que el conjunto de átomos del programa Q es:

*{nat(0),nat(X),nat(s(0)),nat(s(X)),nat(s(s(0))),nat(s(s(X))),. . . }.*

Los átomos del conjunto:

*{nat(0),nat(s(0)), nat(s(s(0))), . . . }.*

son los átomos básicos de Q.

Por tanto, el conjunto de literales de Q es:

*{nat(X),¬nat(X),nat(0),¬nat(0), nat(s(X)), ¬nat(s(X)), nat(s(s(0))), . . . }.*

Y el conjunto de literales básicos:

*{nat(0),¬nat(0), nat(s(0)), ¬nat(s(0)), nat(s(s(0))), . . . }.*

Un programa en ASP consiste en un conjunto de reglas de la forma:

$$L_0 \text{ or } \ldots \text{or } L_i \leftarrow L_{i+1}, \ldots, L_m, \text{not } L_{m+1}, \ldots, \text{not } L_n$$

donde los $L_i$ para $i = 0 \ldots n$ son literales.

El símbolo *not* de la regla es una conectiva que se denomina *negación por defecto*. Esta negación es distinta a la negación clásica, pues en la negación clásica, interpretamos $\neg I$ como "*I es falso*", sin embargo, *not I* debemos interpretarla como "no se sabe si I es cierto"(es decir, $I$ no tiene por qué ser falso). Si consideramos literales $I$ junto a sus negaciones por defecto, *not I*, obtenemos lo que se llama *literales extendidos*.

La conectiva *or* que aparece en la regla es lo que se denomina *disyunción epistémica* y esta disyunción también se diferencia de la clásica. Para ver la diferencia, debemos tener en cuenta que $I_1$ or $I_2$ debe interpretarse como "*se cree que* $I_1$ *es cierto o se cree que* $I_2$ *es cierto*". Así, consideremos $p \vee \neg p$ y $p$ or $\neg p$. La primera expresión es una tautología, es decir, se tiene que o bien p es falso o bien p es cierto (alguna de las dos posibilidades debe satisfascerse). Sin embargo, la segunda la debemos interpretar como "*se cree que p es cierto o se cree que p no es cierto*".

Podemos distinguir dos partes en la regla, separadas por el símbolo $\leftarrow$ :

- La parte de la derecha se denomina *cuerpo* de la regla. Una regla puede tener el cuerpo vacío. En este caso, se denomina *hecho*.
- La parte de la izquierda constituye lo se define como *cabeza* de la regla. Una regla puede no tener cabeza, en cuyo caso se llama *restricción*. Las restricciones son de la forma:

$$\leftarrow L_1, \ldots, L_m, \text{not } L_{m+1}, \ldots, \text{not } L_n$$

**Ejemplo 3.**

Las siguientes reglas definen también un programa P1:

```
const(a).
const(b).
r(a,b).
¬q(a,b).
```

```
q(a,a).
p(a) or p(b) ← q(X,a), ¬q(X,b), not r(X,Y), const(Y),const(X).
```

Tenemos que:

$$\underbrace{\texttt{p(a) or p(b)}}_{cabeza} \leftarrow \underbrace{\texttt{q(X,a),}\neg\texttt{q(X,b),not r(X,Y), const(Y), const(X).}}_{cuerpo}$$

Las cinco primeras reglas son hechos, ya que no poseen cuerpo.

**Ejemplo 4.**

La siguiente regla es una restricción pues no posee cabeza:

```
← nodo(X),nodo(Y), not conectado(X,Y).
```

Se considera que los programas de ASP son un conjunto de reglas *proposicionales.* Las reglas que contienen variables se interpretan como una abreviatura del conjunto de reglas proposicionales que pueden ser obtenidas de la misma sustituyendo las variables por términos básicos.

Dado un programa P de ASP, se denota por *ground(P)* al programa proposicional generado por P.

**Ejemplo 5.**

Sea el programa P1 del **Ejemplo 3**. El programa $ground(P1)$ es:

```
const(a).
const(b).
r(a,b).
¬q(a,b).
q(a,a).
p(a) or p(b) ← q(a,a), ¬q(a,b), not r(a,a), const(a),const(a).
p(a) or p(b) ← q(a,a), ¬q(a,b), not r(a,b), const(b),const(a).
p(a) or p(b) ← q(b,a), ¬q(b,b), not r(b,b), const(b),const(b).
p(a) or p(b) ← q(b,a), ¬q(b,b), not r(b,a), const(a),const(b).
```

Las soluciones de los programas se denominan *conjuntos de respuesta.* Un conjunto de respuesta es un conjunto de literales básicos que constituye el conjunto de certezas que se puede extraer de un programa, cumpliendo:

- Satisface todas las reglas del programa.
- No admite contradicciones.
- Principio de racionalidad: no cree nada que no esté obligado a creer.

Para entender qué es un conjunto de respuesta, debemos aclarar qué significa que un conjunto satisfaga una regla de un programa ASP. Para ello, supongamos que tenemos un conjunto L de literales básicos, entonces:

- Diremos que L safistace el literal $l$ si $l \in$ L. De igual modo, L satisface la negación por defecto del literal, *not* $l$, si $l \notin$ L.

  **Ejemplo 6.** El conjunto $L = \{p(a), \neg p(b)\}$ satisface los literales $p(a)$ y $\neg p(b)$ pues pertenecen al conjunto. También satisface al literal extendido *not* $p(c)$ pues $p(c) \notin L$. El conjunto no satisface el literal *not* $p(a)$ pues $p(a) \in L$.

- Diremos que L satisface $l_0$ or $\ldots$ or $l_n$ disyunción epistémica de literales si $l_i \in$ L para algún $i = 0 \ldots n$.

  **Ejemplo 7.** El conjunto $L = \{p(a), \neg p(b)\}$ satisface *p(a) or p(c)* pues $p(a) \in L$ pero no satisface *¬p(a) or p(c)* porque ni *¬p(a)* ni $p(c)$ pertenecen a $L$.

- Diremos que L satisface un conjunto de literales extendidos R siempre que L satisfaga todos los literales del conjunto R.

  **Ejemplo 8.** El conjunto $L = \{p(a), \neg p(b)\}$ satisface al conjunto $S = \{p(a), not\ p(b)\}$ ya que $p(a) \in L$ y $p(b) \notin L$, pero $L$ no satisface el conjunto $S = \{p(a), p(b)\}$ pues no satisface $p(b)$.

- Diremos que L satisface una regla si L no satisface el cuerpo de la regla o si siempre que L satisfaga el cuerpo de la regla, también satisface la cabeza de la regla.

  **Ejemplo 9.** El conjunto $L = \{p(a), \neg p(b)\}$ :

- Satisface la regla:
  `p(a) or p(b)` ← `p(c), not q(c).`
  pues no satisface el cuerpo.
- Satisface la regla:
  `p(a)` ← ¬`p(b).`
  pues satisface el cuerpo y la cabeza de la regla.
- No satisface:
  `p(b)` ← `p(a), not p(c), not -p(c).`
  pues satisface el cuerpo de la regla y no satisface la cabeza.

Las restricciones, como decíamos antes, son reglas que no poseen cabeza. Por tanto, para que un conjunto L satisfaga una restricción, alguno de los literales extendidos del cuerpo de la regla debe no satisfacerse en L.

**Ejemplo 10.** El conjunto $L = \{p(a)\}$ satisface la restricción:

← `p(a),`¬`q(b),not p(c), not` ¬`q(a).`

### 1.1.1. Semántica informal

La definición que vimos antes de conjunto de respuesta es una definición informal. En esta sección, vamos a estudiar algunos ejemplos de programas y de sus conjuntos de respuestas correspondientes.

**Ejemplo 1.**

`a` ← `b.` % Cree a si cree b
`b.` % Cree b

La segunda regla del programa nos obliga a creer $b$. Como los conjuntos de respuesta del programa deben satisfacer todas las reglas, por la primera regla estamos obligados a creer $a$. Así, el conjunto $\{a\ b\}$ constituye un conjunto de respuesta del programa pues satisface todas las reglas, no contiene contradicciones y verifica el principio de racionalidad. Por otro lado, el conjunto $\{a\ b\ c\}$ verifica también todas las reglas y no contiene contradicciones, pero no es un conjunto de respuesta pues no verifica el principio de racionalidad (no debe creer nada que no esté obligado a creer, y sin embargo no hay ninguna regla que obligue a creer $c$).

**Ejemplo 2** (Negación clásica).

```
¬a ← ¬b.
¬b.
```

Los conjuntos de respuesta del programa deben satisfacer todas las reglas. Para ello, deben contener a $\neg b$ (si no, no se satisfacería la segunda regla). Para satisfacer la primera, como ya satisfacen el cuerpo, deben satisfacer la cabeza, luego también deben contener a $\neg a$. Por tanto, el conjunto $\{\neg b\ \neg a\}$ es un conjunto de respuesta para el programa, ya que satisface las reglas, no contiene contradicciones y verifica el principio de racionalidad.

**Ejemplo 3** (Disyunción epistémica).

```
a or b. %Cree a o cree b.
```

Tanto $\{a\}$ y $\{b\}$ como $\{a\ b\}$ son conjuntos que satisfacen las reglas del programa, pero solo los dos primeros son conjuntos de respuesta pues el tercero no satisface el principio de racionalidad (cree más de lo que está obligado a creer).

Esto no significa que la disyunción epistémica sea excluyente, de hecho el programa:

```
a or b.
a.
b.
```

tiene como conjunto de respuesta a $\{a\ b\}$, lo cual sería una contradicción si la disyunción fuese excluyente (al ser excluyente, se podría creer $a$ o $b$, pero no los dos a vez).

Podemos expresar la disyunción excluyente a través del siguiente programa:

```
a or b.
¬a or ¬b.
```

En cuyo caso los conjuntos de respuesta son $\{a\ \neg b\ \}$ y $\{b\ \neg a\ \}$.

**Ejemplo 4** (Restricciones).

```
a or b ← c.
c.
← a.
```

La segunda regla nos obliga a creer $c$. La primera nos indica que si se cree $c$, se cree $a$ o $b$, luego los conjuntos $\{c\ a\}$ y $\{c\ b\}$ son posibles conjuntos de respuesta. Pero por la tercera nos es imposible creer $a$, luego el único conjunto de respuesta de este programa es $\{c\ b\}$.

**Ejemplo 5** (Negación por defecto).

Supongamos el programa:

```
p(a) ← not p(b).
```

Este programa nos obliga a creer p(a) si p(b) no pertenece al conjunto de certezas. Como no hay ninguna regla que tenga a p(b) en la cabeza, no tenemos por qué creer p(b), luego se verifica el cuerpo de la regla. Para satisfacer la cabeza, se debe creer p(a). De este modo, {p(a)} constituye el único conjunto de respuesta del programa.

Si consideramos ahora el siguiente programa más completo:

```
p(a) ← not p(b).
p(d) ← not p(c).
p(c) ← not p(e).
p(b).
```

La cuarta regla nos obliga a creer p(b). Como se cree p(b), el cuerpo de la primera regla no se verifica. Como no hay ninguna regla que tenga a p(e) en su cabeza, se satisface el cuerpo de la tercera regla, por tanto p(c) formará parte del conjunto de respuesta. Como se cree p(c), no se verifica el cuerpo de la segunda regla. Así, sólo estamos obligados a creer p(b) y p(c). El conjunto de respuesta de este programa es por tanto {p(c) p(b)}.

Pasemos ahora a definir qué se entiende por consecuencia de un programa. Un programa P *implica* un conjunto de literales L si L está contenido en todos los conjuntos de respuesta de P. Se dice entonces que L es una *consecuencia* de P y se denota por $P \models L$.

En la lógica clásica, la relación de implicación tiene la propiedad de ser monótona, es decir, aunque añadamos nuevas hipótesis adicionales a un teorema, las consecuencias del teorema original se mantienen. Sin embargo, la relación de implicación que acabamos de definir no es monótona: al añadir nueva información al programa P, puede ocurrir que las consecuencias de P disminuyan.

Siguiendo con el segundo programa del **Ejemplo 5**, podemos ver que los literales p(b) y p(c) son consecuencias de este, sin embargo, al añadirle una nueva regla:

```
p(a) ← not p(b).
p(d) ← not p(c).
p(c) ← not p(e).
p(b).
p(e).
```

p(c) deja de ser una consecuencia.

Se denomina *consulta* a una conjunción o disyunción de literales. A las consultas que carecen de variables se les llama básicas. La respuesta a una consulta será:

1. Si la consulta es conjuntiva, $L_1 \wedge \ldots \wedge L_n$, entonces:
   - Si P $\models \{L_1,\ldots,L_n\}$, entonces la respuesta es *sí*.
   - Si existe algún $i$ tal que P $\models \overline{L_i}$, entonces la respuesta es *no*.
   - En otro caso, la respuesta es *desconocido*.
2. Si la consulta es disyuntiva, $L_1$ *or*$\ldots$ *or* $L_n$, entonces:
   - Si existe algún $i$ tal que P $\models L_i$, entonces la respuesta es *sí*.
   - Si P $\models \{\overline{L_1},\ldots,\overline{L_n}\}$, entonces la respuesta es *no*.
   - En otro caso, la respuesta es *desconocido*.
3. Si la consulta es de la forma $p(X_1,\ldots,X_n)$ donde los $X_i$ son variables, entonces la respuesta es una lista de términos $t_1,\ldots,t_n$ de manera que P $\models p(t_1,\ldots,t_n)$.

donde $\overline{L}$ denota el literal complementario de $L$.

Consideramos el siguiente ejemplo:

**Ejemplo 6** (Hipótesis del mundo cerrado).

Se considera el programa:

```
p(a) ← not q(a).
```

Como q(a) no aparece como cabeza de ninguna regla, se verifica el cuerpo de la única regla del programa y por tanto el único conjunto de respuesta que posee es {p(a)}. Veamos las respuestas a las siguientes consultas:

- ¿p(a)? La respuesta es *sí*, pues p(a) es consecuencia del programa.
- ¿q(a)? La respuesta es *desconocido*, pues ni q(a) ni ¬q(a) son consecuencias del programa.
- ¿p(a) or q(a)? Como el programa implica p(a), la respuesta es *sí*.
- ¿p(a) ∧ q(a)? La respuesta es *desconocido*, pues p(a) es consecuencia del programa pero ni q(a) ni ¬q(a) son consecuencias.

Consideremos ahora el programa añadiendo una regla más, la cual se denomina *hipótesis del mundo cerrado*:

```
p(a) ← not q(a).
¬q(X) ← not q(X).
```

Debemos entender esta regla como: si q(X) no pertenece al conjunto de certezas del programa entonces q(X) es falso. Esto implica que cada vez que se tenga un término básico $t$, $q(t)$ o $\neg q(t)$ va a pertencer al conjunto de respuesta. Así, el único conjunto de respuesta del programa es {p(a) ¬q(a)} y por tanto las respuestas a las consultas anteriores ahora son:

- ¿p(a)? La respuesta es *sí*, pues p(a) es consecuencia del programa.
- ¿q(a)? La respuesta es *no*, pues ¬q(a) es consecuencia del programa.
- ¿p(a) or q(a)? La respuesta es *sí* porque el programa implica p(a).
- ¿p(a) ∧ q(a)? La respuesta es *no*, pues el programa implica ¬q(a).

### 1.1.2. Semántica formal

Para ver la definición formal de conjunto de respuesta, usamos la noción de *consistencia*: Un conjunto L de literales básicos es *consistente* si no contiene literales complementarios.

Además, debemos tener en cuenta si el programa contiene o no contiene negación por defecto.

**Conjuntos de respuesta I: programas sin negación por defecto.**

Consideremos en primer lugar únicamente programas cuyas reglas no contienen negación por defecto. En estas condiciones, un conjunto L es un *conjunto de respuesta* para un programa P si:

1. L es consistente.
2. L satisface las reglas del programa.
3. L es minimal: ningún subconjunto propio de L satisface todas las reglas de P.

A continuación vamos a ver varios ejemplos, retomando algunos descritos con anterioridad en la Sección 1.1.1:

**Ejemplo 7** (Ejemplo 1 de la Sección 1.1.1)

```
a ← b.
b.
```

El conjunto $L = \{a\ b\}$ es un conjunto de respuesta para este programa pues:

- Es consistente, no contiene literales complementarios.
- Satisface todas reglas del programa: satisface la regla de cuerpo vacío pues $b \in L$. Para satisfacer la primera regla, como satisface el cuerpo de la regla, debe satisfacer la cabeza, lo cual ocurre pues $a \in L$.
- Es minimal: el conjunto $\emptyset$ no satisface la segunda regla. Lo mismo ocurre con {a}. Por otro parte, el conjunto {b} no satisface la primera. De este modo, no hay subconjuntos propios de $L$ que satisfagan todas las reglas.

Además es el único conjunto de respuesta que posee el programa: si existiese otro conjunto $M$ distinto a $L$ de manera que fuese también un conjunto de respuesta del programa, entonces para satisfacer todas las reglas, $L$ tendría que estar contenido en $M$. Pero entonces $L$ sería un subconjunto propio de $M$ que satisfacería todas las reglas del programa, luego $M$ no sería minimal y por tanto llegaríamos a una contradicción.

Algunas observaciones acerca del programa:

- ✧ ¿Es $a$ una consecuencia del programa? *Sí.*
- ✧ ¿Es $b$ una consecuencia del programa? *Sí.*
- ✧ ¿Es $\neg a$ una consecuencia del programa? *No.*
- ✧ ¿Es $\neg b$ una consecuencia del programa? *No.*

**Ejemplo 8** (Ejemplo 3 de la Sección 1.1.1: disyunción epistémica.)

Consideramos de nuevo el programa:

```
a or b.
```

Los conjuntos $L_1 = \{a\}$ y $L_2 = \{b\}$ son conjuntos de respuesta para este programa: satisfacen las reglas, son minimales (el único subconjunto propio que poseen ambos conjuntos es $\emptyset$, que no satisface la regla) y son consistentes. Con esta definición formal de conjunto de respuesta se puede ver también que el conjunto {a b} no es un conjunto de respuesta del programa: No es minimal, pues los subconjuntos {a} y {b} son propios y satisfacen las reglas del programa.

Si nos planteamos algunas de las consultas anteriores, tenemos que:

- ✧ ¿Es $a$ una consecuencia del programa? *Desconocido*, pues para ser consecuencia, $a$ debería pertenecer a todos los conjuntos de respuesta del programa pero $a \notin \{b\}$ , y por otro lado, el programa tampoco implica a $\neg a$
- ✧ ¿Es $b$ una consecuencia del programa? *Desconocido*: al igual que antes, el programa no implica a $b$ pues este no pertenece al conjunto de respuesta {a}, y tampoco implica a $\neg b$.

Otro programa cuyas reglas contienen disyunción epistémica es el siguiente:

```
a or ¬a.
b ← a.
b ← ¬a.
```

Veamos que los conjuntos de respuesta del programa son $L_1$ ={b a} y $L_2$ = {b ¬a}:

1. Ambos conjuntos son consistentes.

2. Verifican las reglas: $L_1$ satisface la primera regla pues contiene al término $a$, satisface la segunda regla pues verifica tanto el cuerpo, ya que contiene al término $a$, como la cabeza, porque contiene a $b$, y satisface la tercera regla al no satisfacer su cuerpo. De manera análoga podemos ver que $L_2$ satisface todas las reglas del programa.

3. Son minimales: todos los términos presentes en $L_1$ son necesarios para satisfacer las reglas del programa, así que este no puede contener subconjuntos propios que verifiquen todas las reglas. Con el mismo razonamiento obtenemos que $L_2$ es minimal.

Como ya comentamos en la Sección 1.1, $a \ or \ \neg a$ no es una tautología. Así, si consideramos el programa anterior sin la primera regla, el único conjunto de respuesta es $\emptyset$, ya que nada nos obliga a creer que $a$ o $\neg a$ sea cierto, luego no se verifica el cuerpo de ninguna de las dos reglas.

**Ejemplo 9.**

Consideremos el siguiente programa P:

```
p(a) ← q(a).
¬p(a).
```

El conjunto $L = \{\neg p(a)\}$ es un conjunto de respuesta del programa pues es consistente, satisface todas las reglas del programa y es minimal (el único subconjunto propio que tiene es el $\emptyset$, que no satisface la segunda regla). De manera análoga que en el ejemplo anterior, se puede ver que de hecho es el único conjunto de respuesta del programa.

Podemos plantear las siguientes consultas:

- ¿¬p(a)? La respuesta es *sí*, pues P $\models \neg$p(a). Por este motivo, la respuesta a la consulta ¿p(a)? es *no*.
- Las consultas ¿q(a)? y ¿¬q(a)? obtienen la misma respuesta: *desconocido*, pues ni q(a) ni su complementario son consecuencias del programa.

*Anotación*: Este ejemplo muestra la diferencia con la implicación clásica. Mientras que en nuestro caso el literal ¬q(a) no es una consecuencia, si hubiésemos utilizado la implicación clásica sí sería una consecuencia.

**Conjuntos de respuesta II: programas con negación por defecto.**

Consideremos ahora programas P cuyas reglas pueden contener negación por defecto. Denotamos por $P^S$ el *programa reducido* de P respecto del conjunto de literales básicos $S$, que se forma eliminando para cada literal $I \in S$ todas las reglas del programa que contienen a *not* $I$ y eliminando de las restantes reglas las premisas que contengan negación por defecto. En esta situación, S es un *conjunto de respuesta* de P si lo es de $ground(P)^S$.

**Ejemplo 10.**

Sea el programa Q:

Regla 1: `p(a) or q(a)` $\leftarrow$ `r(a), not p(b), not q(b).`
Regla 2: `p(a)` $\leftarrow$ $\neg$`r(a), p(b), not q(b).`
Regla 3: `q(a)` $\leftarrow$ `not q(b), not r(b).`
Regla 4: `p(b)` $\leftarrow$ `not q(a), r(b).`
Regla 5: $\neg$`r(b).`
Regla 6: `p(a)` $\leftarrow$ `q(a),` $\neg$`r(b).`

y el conjunto $S = \{r(a), p(b), q(a), \neg r(b)\}$.

Entonces, el programa reducido de Q respecto de S es:

`p(a)` $\leftarrow$ $\neg$`r(a), p(b).` (Proviene de la regla 2)
`q(a).` (Proviene de la regla 3)
$\neg$`r(b).` (Regla 5, no contenía negación por defecto)
`p(a)` $\leftarrow$ `q(a),` $\neg$`r(b).` (Regla 6, no contenía negación por defecto)

La regla 1 del programa Q ha sido eliminada pues contenía al literal extendido *not p(b)* y $p(b) \in S$. Por el mismo motivo se elimina la regla 4, pues aparece en ella *not q(a)* y $q(a) \in S$.

De este modo, para comprobar que S es un conjunto de respuesta de un programa P cualquiera, debemos:

1. Calcular $ground(P)$.
2. Calcular $ground(P)^S$.
3. Comprobar que S es consistente y satisface todas las reglas de $ground(P)^S$.

4. Comprobar que S es minimal, es decir, ningún subconjunto propio de S satisface todas las reglas de $ground(P)^S$.

Proseguimos con algunos ejemplos:

**Ejemplo 11** (Ejemplo 5 de la Sección 1.1.1: negación por defecto.)

Sea P:

```
p(a) ← not p(b).
```

Tomemos el conjunto $S$ ={p(a)}.El programa reducido de P respecto de $S$, $P^S$, es:

```
p(a).
```

ya que como no hay ninguna regla que contenga a *not p(a)*, empezamos directamente a eliminar las premisas que contienen negación por defecto de las reglas. El conjunto S es el único conjunto de respuesta para el programa $P^S$ luego es conjunto de respuesta del programa P.

Veamos ahora el siguiente programa $Q$ del **Ejemplo 5** de la sección 1.1.1:

```
p(a) ← not p(b).
p(d) ← not p(c).
p(c) ← not p(e).
p(b).
```

Sea $L$ ={p(b) p(c)}. Para obtener el programa reducido de $Q$ respecto $L$, eliminamos la primera y la segunda regla, que contienen a los literales extendidos *not p(b)* y *not p(c)*, y después de las reglas restantes eliminamos las premisas que contienen negación por defecto, como *not p(e)*. Así, $Q^L$ tiene la forma:

```
p(c).
p(b).
```

Como el conjunto L es conjunto de respuesta del programa reducido, L es conjunto de respuesta del programa $Q$.

Cabe destacar que algunos programas de ASP no poseen conjuntos de respuesta. A estos programas se les denomina *inconsistentes*.

**Ejemplo 12** (Programa inconsistente).

```
p(a) ← not q(b).
¬p(a).
```

El conjunto $\emptyset$ no satisface la segunda regla, luego no puede ser conjunto de respuesta. El conjunto {¬p(a)} no es conjunto de respuesta pues no verifica la primera regla. El literal q(b) no pertenece al conjunto de certezas. Por tanto, ningún conjunto que lo contenga es conjunto de respuesta ya que no sería minimal. Concluimos que el programa no posee ningún conjunto de respuesta.

Es importante destacar también que un programa ASP puede tener más de un conjunto de respuesta. Veamos un ejemplo.

**Ejemplo 13** (Programa con dos conjuntos de respuesta).

Consideremos el siguiente programa:

```
q(a) ← not p(a).
p(a) ← not q(a).
```

Es fácil comprobar que este programa P tiene dos conjuntos de respuesta S1={p(a)} y S2={q(a)}.

Por último, veamos algunas propiedades que poseen los conjuntos de respuesta de un programa $P$:

1. Supongamos que $P$ no contiene variables. Entonces, si $S$ es un conjunto de respuesta:
   - $S$ satisface todas las reglas de $P$, es consistente y minimal.
   - Para cada literal I presente en el conjunto de respuesta, debe existir una regla r en $P$ de manera que el conjunto de respuesta satisface su cuerpo y el literal I es el único literal en la cabeza de la regla que satisface $S$. De este modo, se dice que la regla r *sustenta* al literal I.
2. Sea $P = Q \cup R$, donde $R$ es el conjunto de todas las restricciones de $P$. Entonces, los conjuntos de respuesta de $P$ son los conjuntos de respuesta de $Q$ que satisfacen todas las restricciones de $R$.

**Ejemplo 14** (Gelfond&Lifschitz '1988).

El ejemplo que vamos a exponer a continuación es el que se usó en el artículo [GL88] donde por primera vez se introduce la noción de conjunto de respuesta.

Consideremos el programa $P$:

```
p(1,2).
q(X) ← p(X,Y),not q(Y).
```

El programa $ground(P)$ es el siguiente:

```
p(1,2).
q(1) ← p(1,1),not q(1).
q(1) ← p(1,2),not q(2).
q(2) ← p(2,1),not q(1).
q(2) ← p(2,2),not q(2).
```

Sea el conjunto $S = \{p(1,2), q(1)\}$, calculemos $ground(P)^S$:

```
p(1,2).
q(1) ← p(1,2).
q(2) ← p(2,2).
```

El conjunto $S$ es un conjunto de respuesta para $ground(P)^S$, pues verifica las reglas, es consistente y es minimal. Entonces, $S$ es conjunto de respuesta de $P$. Podemos observar que $S$ es un conjunto sustentado de $P$ pues *p(1,2)* está sustentado por la primera regla de $ground(P)$ y *q(1)* está sustentado por la tercera regla de $ground(P)$.

## 1.2. Sintaxis del sistema CLINGO

### 1.2.1. Algunas generalidades

En CLINGO, se usa el símbolo **:-** en lugar de ← para separar el cuerpo y la cabeza de las reglas. Además, todas las reglas deben terminar con un punto final. Así, una regla en CLINGO tiene la forma:

$$A_0, \ldots, A_n \text{ :- } I_0, \ldots, I_m.$$

donde los literales $A_i$ para todo i = 0,. . . ,n constituyen la cabeza de la regla y los literales $I_i$ para todo i = 0,. . . ,m constituyen el cuerpo.

Los nombres de predicados, funciones o constantes deben empezar por letra minúscula y los nombres de variables, por letra mayúscula. Las *variables anónimas* en CLINGO se pueden denotar por `"_"`. Una misma variable suele aparecer más de una vez en las reglas. Si la variable sólo aparece una vez en la regla, no es neceario asignarle un nombre, y para este tipo de casos se usan las variables anónimas. Por ejemplo, en el programa:

```
datos(maria,15,sevilla).
datos(ana,17,malaga).
datos(juan,25,sevilla).
datos(ivan,30,huelva).
sevillano(X) :- datos(X,C1,sevilla).
```

La variable `X` aparece dos veces en la regla, nos indica que la persona `X` es sevillana si en sus datos aparece como primer argumento `X` y como tercer argumento `sevilla`. Sin embargo, la variable `C1`, que aparece sólo una vez en la regla y que representa la edad de la persona, no nos interesa, por eso podemos reemplazarla por una variable anónima. De esta manera, el programa quedaría igual, sustituyendo la última línea por:

```
sevillano(X) :- datos(X,_,sevilla).
```

También podemos utilizar *tuplas*. En las tuplas, los argumentos se encuentran entre paréntesis y separados por comas.

Denominamos *instancias* de una regla a las particularizaciones de la misma que se obtienen sustituyendo las variables de la regla por constantes.

Como ejemplo, consideremos el siguiente código:

```
clase(juan,1).
clase(antonio,2).
clase(daniel,2).
clase(ana,1).

amigos(X,Y) :- clase(X,A), clase(Y,B), A=B, X!=Y.
```

Las primeras 4 líneas del código indican en que clase de un colegio (1 o 2) se encuentra cada alumno (Juan, Antonio, Daniel, Ana). La quinta línea es una regla, que se interpreta como " X es amigo de Y si la clase de X es la misma que la de Y y X e Y no son la misma persona ".

Una instancia de la regla es la siguiente:

```
amigos(juan,ana) :- clase(juan,1), clase(ana,1), 1=1, juan != ana.
```

### 1.2.2. Disyunción

Para escribir en CLINGO la disyunción epistémica, se usa el símbolo "`;`", de manera que las reglas tienen la forma:

$$L_0; \ldots; L_m \ \texttt{:-}\ L_{m+1}, \ldots, L_n.$$

Para que se satisfaga la cabeza de la regla, al menos uno de los literales $L_j$, con j = 0,. . . ,m, debe ser cierto.

Por ejemplo, el programa:

```
p(a);p(b).
```

tiene dos soluciones, el conjunto {p(a)} y el conjunto {p(b)}.

El conjunto {p(a) p(b)} no es una solución por el principio de racionalidad.

*Nota:* También se usa el símbolo "|" para representar la disyunción epistémica.

### 1.2.3. Constantes boolenas

Las constantes booleanas son los literales `#true` y `#false`, que se interpretan en cada aparición como verdadero o falso respectivamente. De esta forma, con el programa:

```
#true.
```

CLINGO nos indica que hay una solución, pues el valor de `#true` es "verdad", luego se satisface trivialmente, mientras que el programa:

```
#false.
```

no tiene ninguna solución pues el valor de `#false` es "falso", de nuevo es trivial el hecho de que no hay ningún conjunto que pueda verificarlo.

### 1.2.4. Funciones aritméticas

En CLINGO podemos realizar operaciones aritméticas utilizando los símbolos:

- Suma y resta: `+` y `-`, respectivamente.
- Multiplicación: $*$
- Exponenciación: $**$
- Módulo: $\backslash$
- División entera: /
- Valor absoluto: `||`

Como ejemplo, tomemos el programa:

```
abs(|-6|).
suma(10+2).
eleva(2**3).
```

Si ejecutamos este programa en CLINGO (hemos guardado el programa en el fichero con nombre Ejemplo_Aritmetica, que termina en la extensión lp para que CLINGO pueda leer el archivo) este nos devuelve el modelo:

```
% clingo version 4.5.4
% Reading from Ejemplo_Aritmetica.lp
% Solving...
% Answer: 1
% abs(6) suma(12) eleva(8)
% SATISFIABLE
```

### 1.2.5. Predicados de igualdad y orden

A través de los símbolos:

$$= \quad != \quad < \quad > \quad <= \quad >=$$

podemos realizar comparaciones entre enteros, constantes y términos. Para realizar una comparación entre enteros, CLINGO tiene en cuenta el orden usual de los enteros, mientras que para realizar una comparación entre constantes, tiene en cuenta el orden lexicográfico. Los enteros en CLINGO son menores que las constantes, que a su vez son menores que los términos.

Para ilustrar como CLINGO realiza las comparaciones, tomemos el programa:

```
constante(a). constante(ab). constante(ac).
entero(2). entero(3). entero(1).
es_mayor(X,Y) :- X>Y, constante(X), constante(Y).
es_mayor_enteros(X,Y) :- X>Y, entero(X), entero(Y).
suma_mas_de_tres(X,Y) :- X+Y >5, entero(X), entero(Y).
```

CLINGO nos da como solución a la primera regla:

%es_mayor(ab,a) es_mayor(ac,a) es_mayor(ac,ab)

La solución que nos muestra para la segunda regla es:

%es_mayor_enteros(3,2) es_mayor_enteros(2,1) es_mayor_enteros(3,1)

Comprobamos así que CLINGO toma el orden usual para realizar comparaciones entre enteros.

Como solución a la tercera y última regla, obtenemos:

%suma_mas_de_tres(3,3)

Esto nos muestra que primero realiza la operación aritmética y luego compara el resultado de esta con el entero 5.

El símbolo "="también puede utilizarse para abreviar. Por ejemplo, en la regla:

```
prod_9(X) :- X*X=Z, Z=9, entero(X).
```

usamos el primer igual para abreviar el producto de la variable X consigo misma por Z y el segundo lo usamos para realizar una comparación entre Z y 9.

### 1.2.6. Intervalos

Para construir intervalos, se usan expresiones de la forma $s..r$, donde suponemos que $s <= r$ , que representan la sucesión de números naturales consecutivos entre s y r. Vemos un ejemplo de intervalos en un código:

```
pares_iguales(X,Y) :- X=0..2, Y=0..2, X=Y.
```

Obtenemos los pares (X,X) con X entre 0 y 2.

En este ejemplo, los intervalos se encuentran en el cuerpo de la regla por lo que se expanden de forma disyuntiva. Si el intervalo aparece en la cabeza de la regla, se expande de forma conjuntiva. Por ejemplo, el hecho:

```
p(2..5).
```

se expande al conjunto de hechos:

```
p(2).
p(3).
p(4).
p(5).
```

### 1.2.7. Agrupación

En los programas puede aparecer el mismo símbolo de predicado o función varias veces evaluado en distintos argumentos. Este es el caso del ejemplo que se expone en la sección 1.2.5:

```
constante(a). constante(ab). constante(ac).
entero(2). entero(3). entero(1).
es_mayor(X,Y) :- X>Y, constante(X), constante(Y).
```

En este ejemplo, los predicados `constante()` o `entero()` aparecen más de una vez en el programa evaluados en argumentos diferentes.

La agrupación se usa para escribir de forma compacta los argumentos de predicados o funciones, evitando así el uso reiterado de estos. Para ello, separamos los argumentos con ";". De esta forma, el código anterior puede reescribirse como:

```
constante(a;ab;ac).
entero(2;3;1).
es_mayor(X,Y) :- X>Y, constante(X), constante(Y).
es_mayor_enteros(X,Y) :- X>Y, entero(X), entero(Y).
```

Como ocurría en el caso de los intervalos, dependiendo de si la agrupación se encuentra en la cabeza o en el cuerpo de la regla, se expande de forma conjuntiva o disyuntiva, respectivamente.

### 1.2.8. Condicionales

Para escribir condicionales en CLINGO, se usa el símbolo ":". Los condicionales son de la forma:

$$L_0 : L_1, \ldots, L_n$$

donde los $L_j$ para todo j=0 ... n son literales. A $L_1, \ldots, L_n$ se le denomina *condición*. Los literales que forman la condición se encuentran separados por comas, al igual que los literales que aparecen en el cuerpo de las reglas. Para no confundir los literales que aparecen después de un condicional con los literales que constituyen la condición, se usa el símbolo ";" al final de la condición. Esto es, supongamos que tenemos el programa:

```
elemento(1..4).
consecutivos(X,Z):- elemento(X), #false : elemento(Y), X<Y, Y<Z;
                                  elemento(Z), X<Z.
```

Hemos usado ";" para evitar la confusión entre los literales que pertenecen a la condición del condicional y los que no. El condicional en esta regla evita que se verifique el predicado `consecutivos(X,Z)` cuando existe un número intermedio entre X y Z.

En las reglas de los programas aparecen variables que no se encuentran sujetas a ningún condicional. Estas variables se denominan *globales* y en el proceso de instanciación, se sustituyen por términos antes que las variables que sí están sujetas a condicionales. Es por esta razón que no debemos nombrar a las variables que aparecen en las expresiones condicionales de forma que se puedan confundir con las globales.

### 1.2.9. Restricciones

Las restricciones son expresiones de la forma:

$$s_1 <_1 \alpha\{t_1 : L_1; \ldots; t_n : L_n\} <_2 s_2$$

donde:

- Los elementos $t_i$ y $L_i$ son tuplas de términos y literales respectivamente. Si alguna de las tuplas de literales está vacía, sólo aparece la tupla de términos que la precede (suponemos que no está vacía) y no aparecen los dos puntos.
- $s_1$ y $s_2$ son términos y $<_1$ y $<_2$ son predicados de comparación.
- $\alpha$ es una función que se aplica a las tuplas de términos que se encuentran dentro de las llaves una vez se hayan evaluado los condicionales.

CLINGO trata a los elementos que aparecen entre las llaves como elementos de un conjunto. Por ello, si aparece un condicional repetido dentro de las llaves, se trata como un único elemento.

Los predicados $<_1$ y $<_2$ pueden ser reemplazados por "<=", en cuyo caso obtenemos una cota superior o inferior para el conjunto.

$\alpha$ puede ser alguna de las funciones siguientes:

- #count, que nos da el número de elementos que posee el conjunto.
- #sum: Suma de los pesos de las tuplas de términos del conjunto (con peso, nos referimos al primer elemento de la tupla). También $\alpha$ puede ser #sum+, en cuyo caso se suman solo los pesos positivos.
- #min y #max, que nos dan el mínimo elemento y el máximo del conjunto, respectivamente.

Veamos ahora un ejemplo. Supongamos que queremos ver las distintas formas de invertir en 4 empresas de manera que obtengamos mínimo 1000 euros. Esto lo podemos escribir en CLINGO a través de la restricción:

```
1000 <= #sum { 300 : empresa_1; 250: empresa_2; 600: empresa_3;
                250: empresa_4}.
```

Si se decide invertir en la empresa 1 y en la empresa 3, el conjunto se reduce a {300,600}. Como al aplicar la suma obtenemos 900, no se verifica el predicado de comparación y por tanto no se verifica la restricción, luego no es una posible solución. Para que CLINGO nos de todas las soluciones, al llamarlo debemos incluir un cero (supongamos que hemos llamado al archivo donde se encuentra el programa Ejemplo_Restricciones):

% clingo Ejemplo_Restricciones.lp 0

Esto nos da:

% Reading from Ejemplo_Restricciones.lp
% Solving...
% Answer: 1
% empresa_1 empresa_4 empresa_3
% Answer: 2
% empresa_1 empresa_2 empresa_3
% Answer: 3
% empresa_1 empresa_2 empresa_4 empresa_3

# Capítulo 2

# Propiedades de los programas en ASP

A lo largo de este capítulo, vamos a exponer algunas de las propiedades que poseen los programas de ASP. Además, estudiaremos cuándo un programa de ASP es consistente, es decir, posee algún conjunto de respuesta, y los requisitos bajos los cuales este conjunto de respuesta es único.

## 2.1. Conjuntos de respuesta: Existencia, unicidad y propiedades

Consideremos un programa P en ASP compuesto por reglas de la forma:

$$A_0 \ or \ \ldots \ or \ A_i \leftarrow A_{i+1}, \ldots, A_m, not \ A_{m+1}, \ldots, not \ A_n$$

donde los $A_i$ para $i = 0 \ldots n$ son literales.

Sean $L_1$ y $L_2$ dos conjuntos de respuesta distintos de P. Entonces, se verifica que $L_1 \not\subseteq L_2$ y $L_2 \not\subseteq L_1$ (los conjuntos no pueden compararse respecto a la inclusión).

**Observación**: Si un conjunto unitario, por ejemplo $L = \{a\}$, es un conjunto de respuesta para P, no pueden existir otros conjuntos de respuesta de P que contengan al término $a$, pues si existiese un conjunto $M$ tal que $a \in M$ entonces $L \subseteq M$ y esto no puede ocurrir como acabamos de ver. Por el mismo razonamiento, si $\emptyset$ es un conjunto de respuesta de P, entonces es el único

conjunto de respuesta de P pues todo conjunto $L$ verifica que $\emptyset \subseteq L$.

Consideremos ahora programas de ASP cuyas reglas no contienen negación clásica ni restricciones, es decir, programas que solo poseen reglas de la forma:

$$A_0 \ or \ \ldots \ or \ A_i \leftarrow A_{i+1}, \ldots, A_j, \ not \ A_{j+1}, \ldots, \ not \ A_m. \tag{2.1}$$

donde los $A_s$ son átomos para todo $s \geq 0$. Este tipo de programas se denominan *normales*.

**Ejemplo 1.**

El siguiente programa de una regla:

```
p(a) ← not p(a).
```

es un programa normal, ya que no posee restricciones ni negación clásica y además, es inconsistente.

Con esto, vemos que no podemos asegurar la existencia de conjuntos de respuesta para programas normales. Sin embargo, para algunos tipos de programas normales podremos garantizar no sólo la existencia, sino también la unicidad de conjuntos de respuesta. En primer lugar, estudiemos los denominados *programas positivos*.

**Definición 1 (Programas positivos)** *Se dice que un programa $P$ es **positivo** si sus reglas no poseen negación por defecto, es decir, si sus reglas tienen la forma:*

$$A_0 \ or \ \ldots \ or \ A_i \leftarrow A_{i+1}, \ldots, A_j. \tag{2.2}$$

*donde los $A_s$ son átomos y $s \geq 0$.*

**Ejemplo 2.**

El siguiente programa es un programa positivo:

```
estudiante(juan,colegio).
estudiante(ana,univ).
estudiante(marta, instituto).
```

```
estudiante(marco, univ).
universitario(X) ← estudiante(X, univ).
```

Para estudiar si este programa posee algún conjunto de respuesta, usamos la proposición:

**Proposición 1** *Sea P un programa positivo. Entonces, P es consistente. Más aún, si las reglas de P no contienen disyunción, entonces P posee un único conjunto de respuesta.*

Como el programa anterior no posee disyunción en sus reglas, usando la proposición que acabamos de ver podemos asegurar que el programa posee un único conjunto de respuesta.

**Ejemplo 3.**

En el caso del programa (lo escribimos con la sintaxis de CLINGO para poder utilizarlo):

```
estudiante(ana;juan).
colegio(X);universidad(X) :- estudiante(X).
```

sabemos que tiene al menos un conjunto de respuesta porque es positivo, pero como posee disyunción en sus reglas, no podemos asegurar que posea un único conjunto de respuesta. De hecho, CLINGO nos devuelve cuatro conjuntos de respuesta para este programa:

% Answer: 1
% estudiante(ana) estudiante(juan) colegio(ana) universidad(juan)
% Answer: 2
% estudiante(ana) estudiante(juan) colegio(ana) colegio(juan)
% Answer: 3
% estudiante(ana) estudiante(juan) universidad(ana) universidad(juan)
% Answer: 4
% estudiante(ana) estudiante(juan) universidad(ana) colegio(juan)

**Ejemplo 4.**

Sea el programa:

```
r(a,b).
r(b,c).
```

```
r(Y,X) ← r(X,Y).
r(X,Z) ← r(X,Y),r(Y,Z).
```

Este programa es positivo y por lo tanto es consistente. Como no posee disyunción epistémica en las reglas, podemos asegurar también que el programa tiene un único conjunto de respuesta.

Obsérvese que el anterior programa positivo construye el cierre simétrico y transitivo de la relación binaria **r/2**.

Veamos ahora las nociones de *programas estratificados* y *localmente estratificados*, para los que también vamos a poder exponer algunos resultados de existencia y unidad de conjuntos de respuesta.

**Definición 2 (Grafo de dependencia)** *Consideremos un programa $P$ cuyas reglas son de la forma* (2.1)*. Sea $Q$ el conjunto formado por los nombres de predicados del programa $P$. Se denomina* ***grafo de dependencia*** *de $P$ al grafo cuyos nodos son los símbolos de predicado presentes en $Q$ y cuyas aristas son de la forma $(p_1, p_2, +)$ o $(p_1, p_2, -)$, donde $p_1$ y $p_2$ son símbolos de predicado, que se denominan respectivamente arco positivo y negativo y que aparecen en el grafo según:*

1. *Si existe una regla $r$ en $P$ verificando que su cabeza contiene un átomo formado por el símbolo de predicado $p_1$ y su cuerpo contiene un átomo formado por $p_2$, entonces el arco $(p_1, p_2, +)$ aparece en el grafo.*
2. *Si existe una regla $r$ en $P$ verificando que su cabeza contiene un átomo formado por el símbolo de predicado $p_1$ y su cuerpo contiene un literal de la forma not I, con I el átomo formado por el símbolo de predicado $p_2$, entonces el arco $(p_1, p_2, -)$ aparece en el grafo.*

**Definición 3 (Función de nivel)** *Sea $P$ un programa normal sin variables. Las funciones $||\ ||$ que asocian a cada elemento del conjunto de los átomos básicos de $P$ un número natural se denominan* ***funciones de nivel*** *para $P$.*

*Observaciones:*

1. Dos nodos $p_1$ y $p_2$ de un mismo grafo de dependencia pueden aparecer conectados por un arco positivo $(p_1, p_2, +)$ y uno negativo $(p_1, p_2, -)$ simultáneamente.

2. Sea $D = p_1$ *or* ... *or* $p_n$ una disyunción epistémica de atómos de un programa P y sea $||\ ||$ la función de nivel para el programa. Entonces, $||D|| = min\{||p_1||, \ldots, ||p_n||\}$.

**Ejemplo 5.**

Sea el programa normal:

```
p(a) or p(c) ← not p(b).
p(c) ← p(d), not p(a).
p(d).
```

El conjunto de átomos básicos del programa es {*p(a),p(b),p(c),p(d)*}.

Tenemos que la siguiente función es un ejemplo de función de nivel para el programa P:

$||p(a)|| = 2$, $||p(b)|| = 2$, $||p(c)|| = 3$, $||p(d)|| = 0$

**Definición 4 (Programas estratificados)** *Sea P un programa de la forma* (2.1) *y sea G su grafo de dependencia. Se dice que P es un* ***programa estratificado*** *si G no contiene ningún ciclo negativo, esto es, ciclos que contienen al menos un arco negativo.*

**Definición 5 (Programas localmente estratificados)** *Sea un programa P cuyas reglas son de la forma:*

$$A_0 \text{ or } \ldots \text{or } A_i \leftarrow A_{i+1}, \ldots, A_j, \text{ not } A_{j+1}, \ldots, \text{ not } A_m.$$

*y sea ground(P) el programa formado por reglas r de la forma:*

$$r\text{: } A_0 \text{ or } \ldots \text{or } A_i \leftarrow A_{i+1}, \ldots, A_j, \text{ not } A_{j+1}, \ldots, \text{ not } A_m.$$

*Entonces se dice que P está* ***localmente estratificado*** *si existe una función de nivel $||\ ||$ para ground(P) de manera que para cada regla r del programa ground(P), se verifica:*

1. $||A_k|| \leq ||cabeza(r)||$ *para todo* $A_k$ *con* $i < k \leq j$.
2. $||A_k|| < ||cabeza(r)||$ *para todo* $A_k$ *con* $j < k \leq m$.

Sigamos con los ejemplos.

**Ejemplo 6.**

Consideramos el programa R:

```
p(a) ← q(a).
t(a) ← not q(b).
q(a).
```

Los nodos del grafo de dependencia de R son $p$, $q$ y $t$ .
El arco $(p, q, +)$ debe aparecer en el grafo porque en la primera regla se encuentra $p(a)$ en la cabeza y $q(a)$ en el cuerpo.
Del mismo modo, el arco $(t, q, -)$ debe aparecer en el grafo pues en la cabeza de la segunda regla aparece *t(a)* y en su cuerpo, *not q(b).*
Como no contiene ciclos negativos, el programa R está estratificado.

**Ejemplo 7.**

Consideremos el programa S:

```
q(b) ← q(a).
q(a).
```

Las funciones:

$$||q(s)||_1 = \begin{cases} 1 & \text{si } s = b \\ 0 & \text{si } s = a \end{cases} \qquad ||q(s)||_2 = \begin{cases} 0 & \text{si } s = b \\ 1 & \text{si } s = a \end{cases}$$

son funciones de nivel para el programa S. Sin embargo, $||\ ||_2$ no verifica las condiciones de la Definición 5, pues para la regla 1 no se verifica $||q(a)||_2 = 1 \leq ||q(b)||_2 = 0$. Aún así, se tiene que S es un programa localmente estratificado pues $||\ ||_1$ si verifica las condiciones de la definición.

*Observaciones:*

1. Los programas que no contienen negación clásica ni negación por defecto están localmente estratificados (una función de nivel $||\ ||$ para estos programas que verifica las condiciones de la Defínción 5 es la función que asocia cada átomo $p_k$ que aparece en las reglas con el número 0, es decir, $||p_k|| = 0$). En este caso se encuentra el programa S que acabamos de ver en los ejemplos.

2. Si un programa está estratificado, entonces también está localmente estratificado; sin embargo, el recíproco no es cierto. El programa R

anterior está estratificado, luego también está localmente estratificado. Un ejemplo sencillo de programa P que está localmente estratificado y no es estratificado es el siguiente:

`p(a)` $\leftarrow$ `not p(b).`

Es claro que P no está estratificado porque su grafo de dependencia contiene el arco negativo $(p, p, -)$ y, por tanto, contiene ciclos negativos. Sin embargo, P sí es localmente estratificado. Basta considerar la función de nivel dada por $||p(a)|| = 1$ y $||p(b)|| = 0$. De hecho, P tiene un único conjunto de respuesta S={p(a)}.

La siguiente proposición nos aporta resultados de existencia y unicidad de los conjuntos de respuesta de programas estratificados y localmente estratificados:

**Proposición 2** *Sea P un programa de la forma* (2.1)*. Se verifica:*

1. *Si el grafo de dependencia de P no posee ciclos con un número impar de arcos negativos, entonces P es consistente.*
2. *Si P que está localmente estratificado, es consistente.*
3. *Si P está estratificado y sus reglas no poseen disyunción, entonces P posee un único conjunto de respuesta.*
4. *Si P está localmente estratificado y sus reglas no poseen disyunción, entonces posee un único conjunto de respuesta (condición suficiente pero no necesaria).*

*Anotaciones:*

- El programa R que vimos en el ejemplo 1 está estratificado y no contiene disyunciones, luego sabemos por la proposición que acabamos de ver que posee un único conjunto de respuesta. De manera análoga, como el programa S del ejemplo 2 está localmente estratificado y no contiene disyunciones, podemos concluir que posee un único conjunto de respuesta.
- Los puntos 2 y 4 de la proposición anterior también se verifican cuando añadimos a programas que están localmente estratificados reglas como la *hipótesis del mundo cerrado*, es decir, reglas de la forma:

  $\neg$`q(X)` $\leftarrow$ `not q(X).`

Veamos por último algunos ejemplos más.

**Ejemplo 8.**

El programa Q:

```
p(a) ← not q(a).
q(a) ← not p(a).
```

y el programa P:

```
p(a) ← not q(a).
q(a) ← not r(a).
r(a) ← not p(a).
```

no son estratificados, pues el grafo de dependencia del primer programa contiene al ciclo negativo formado por los arcos $(p, q, -)$ y $(q, p, -)$ y el grafo de dependencia del segundo contiene al ciclo negativo formado por los arcos $(p, q, -)$, $(q, r, -)$ y $(r, p, -)$.

Además, estos programas se comportan de forma distinta: mientras que P es consistente (tiene dos conjuntos de respuesta $\{p(a)\}$ y $\{q(a)\}$), el programa Q es inconsistente (no posee ningún conjunto de respuesta). El apartado 1 de la proposición anterior justifica estas diferencias: como acabamos de ver, el grafo de dependencia de P contiene un ciclo con dos arcos negativos luego podemos aplicar el primer apartado de la proposición anterior y asegurar que P es consistente; sin embargo, el grafo de dependencia del programa Q contiene un ciclo con 3 arcos negativos, luego el apartado 1 de la proposición anterior no es aplicable a este programa.

Que el grafo de dependencia de un programa $R$ no contenga ciclos con un número impar de arcos negativos nos da una condición suficiente para la consistencia, pero no necesaria. De hecho, el programa P que vimos anteriormente:

```
p(a) ← not p(b).
```

verifica que su grafo de dependencia contiene un ciclo negativo de longitud 1 y sí tiene conjuntos de respuesta.

**Ejemplo 9.**

Sea el programa S:

```
seta(a).
seta(b).
gris(a) ← not rojo(a).
rojo(b) ← not gris(b).
venenosa(X) ← seta(X),rojo(X).
comestible(X) ← seta(X), not venenosa(X).
```

Es claro que el programa S es normal. Calculemos ahora *ground(S)*:

H1: `seta(a).`
H2: `seta(b).`
R1: `gris(a) ← not rojo(a).`
R2: `rojo(b) ← not gris(b).`
R3: `venenosa(a) ← seta(a),rojo(a).`
R4: `venenosa(b) ← seta(b),rojo(b).`
R5: `comestible(a) ← seta(a), not venenosa(a).`
R6: `comestible(b) ← seta(b), not venenosa(b).`

Demos una función de nivel para $ground(S)$ que muestre que S está localmente estratificado.

Los átomos del programa son:

{*seta(a),seta(b),rojo(a),rojo(b),gris(a),gris(b),venenosa(a),*

*venenosa(b),comestible(a),comestible(b)*}

Para el cálculo de la función de nivel || ||, tomamos:

$||seta(a)|| = v_1 \quad ||seta(b)|| = v_2 \quad ||rojo(a)|| = v_3 \quad ||rojo(b)|| = v_4$

$||gris(a)|| = v_5 \quad ||gris(b)|| = v_6 \quad ||venenosa(a)|| = v_7$

$||venenosa(b)|| = v_8 \quad ||comestible(a)|| = v_9 \quad ||comestible(b)|| = v_{10}$

Obtenemos un sistema de inecuaciones aplicando la definición 5:

Por R1: $v_3 < v_5$
Por R2: $v_6 < v_4$
Por R3: $v_1 <= v_7$ && $v_3 <= v_7$
Por R4: $v_2 <= v_8$ && $v_4 <= v_8$

Por R5: $v_1 <= v_9$ && $v_7 < v_9$
Por R6: $v_2 <= v_{10}$ && $v_8 < v_{10}$

Una solución de este sistema de inecuaciones viene dado, por ejemplo, por:

$v1 = v2 = 0$ (ni seta(a) ni seta(b) aparecen en la cabeza de ninguna regla que no sea un hecho)

$v3 = v6 = 0$ (ni rojo(a) ni gris(b) aparecen en la cabeza de ninguna regla)

$v4 = v5 = 1$ (para satisfacer R1 y R2)

$v7 = v8 = 2$ (para satisfacer R3 y R4)

$v9 = v10 = 3$ (para satisfacer R5 y R6)

Luego, la siguiente función es una función de nivel para $ground(S)$:

$||seta(a)|| = 0 \quad ||seta(b)|| = 0 \quad ||rojo(a)|| = 0 \quad ||rojo(b)|| = 1$

$||gris(a)|| = 1 \quad ||gris(b)|| = 0 \quad ||venenosa(a)|| = 2$

$||venenosa(b)|| = 2 \quad ||comestible(a)|| = 3 \quad ||comestible(b)|| = 3$

y esto nos permite demostrar que S está localmente estratificado. Puesto que en S no aparece la disyunción epistémica, por la proposición 2 podemos inferir que S es consistente y tiene un único conjunto de respuestas. De hecho, dicho conjunto de respuestas viene dado por:

$\{rojo(b)\ gris(a)\ seta(a)\ seta(b)\ venenosa(b)\ comestible(a)\}$

# Capítulo 3

# Bases de conocimiento

En este capítulo vamos a centrarnos en cómo construir las denominadas *bases de conocimiento*, las cuales nos sirven para modelizar el contexto que queremos estudiar y así poder razonar sobre él, exponiendo ejemplos sobre algunas bases.

## 3.1. Introducción

Un *agente inteligente* es una entidad que observa su entorno y actúa (de manera no trivial) para conseguir sus objetivos. El conjunto de reglas y hechos que se usan para modelizar el entorno sobre el que queremos que razone el agente se denomina *base de conocimiento*.

Para la creación de la base de conocimiento, debemos tener en cuenta que:

1. Modelizaremos contextos reales, de los cuales tendremos, además de la información correspondiente, la que proviene del sentido común. En algunas situaciones, modelizaremos este conocimiento para que el agente pueda razonar correctamente sobre el contexto.
2. Mucha información acerca del entorno será modelizada a través de definiciones recursivas y usando una organización jerárquica del conocimiento.
3. En determinadas ocasiones, completaremos la información que estamos

representando usando, por ejemplo, reglas de la forma de la hipótesis del mundo cerrado.

Para aclarar estas nociones, veamos algunas bases de conocimiento.

## 3.2. Modelización de un barrio

Queremos modelizar las relaciones entre las personas que viven en un barrio. Supongamos que tenemos un barrio B y conocemos la siguiente información:

1. Marta, Julia, Manuel y Rocío viven en el barrio B.
2. Julia vive con Marta y Manuel vive con Rocío.

Para la modelización, usamos el predicado **persona/1** para clasificar a las personas que viven en el vecindario:

```
persona(julia;marta;manuel;rocio).
```

Utilizamos el predicado **convivientes/2** para expresar la relación "vivir en la misma casa" y el predicado **residencia_B/1** para expresar la relación "vivir en el barrio B":

```
residencia_B(julia;marta;manuel;rocio).
convivientes(julia,marta;manuel,rocio).
```

*Nota: Estamos usando la sintaxis de CLINGO. Se recuerda que ";" se usa para la agrupación.*

**Concepto: Vecinos.**

Nuestro objetivo es obtener información nueva acerca de quiénes son vecinos.

Dos personas serán vecinas si viven en el mismo barrio (en este caso, si viven en B) y no consta que sean convivientes.

Un primer intento de definir la relación es utilizar la regla:

```
vecinos(X,Y) :- residencia_B(X), residencia_B(Y),
                not convivientes(X,Y).
```

Sin embargo, al ejecutar el código (las tres reglas conjuntamente) obtenemos:

% Answer: 1
% **vecinos(julia,julia) vecinos(marta,julia)** vecinos(manuel,julia)
% vecinos(rocio,julia) **vecinos(marta,marta)** vecinos(manuel,marta)
% vecinos(rocio,marta) vecinos(julia,manuel) vecinos(marta,manuel)
% **vecinos(manuel,manuel) vecinos(rocio,manuel)** vecinos(julia,rocio)
% vecinos(marta,rocio) **vecinos(rocio,rocio)**

*Nota: hemos usado la directriz #show para que sólo salgan los hechos relacionados con el predicado vecinos.*

Los hechos recalcados no deberían aparecer en el conjunto de respuesta ya que una persona no puede ser su propio vecino y además tanto Rocío y Manuel como Marta y Julia son convivientes, luego carece de sentido que sean a su vez vecinos.

El problema reside en que:

1. No hemos modelizado el conocimiento ímplicito de que una persona no puede ser vecina de sí misma.
2. No hemos modelizado lo que sabemos por sentido común: la relación "ser conviviente" es una relación simétrica, es decir, si a es conviviente con b, b es conviviente con a, por tanto, ni a puede ser vecino de b, ni b de a.

### Carácter simétrico de la relación convivientes/2.

Añadimos la siguiente regla al programa, que representa el carácter simétrico de la relación *convivientes*:

```
convivientes(X,Y) :- convivientes(Y,X).
```

Veamos un segundo intento (ya definitivo) de representación de la relación vecinos. Definimos **vecinos/2** de la siguiente forma: X e Y son vecinos si ambos residen en el barrio B, no son la misma persona y no consta que sean convivientes. De esta manera, la regla queda como:

```
vecinos(X,Y) :- residencia_B(X), residencia_B(Y),
                not convivientes(X,Y), X != Y.
```

y obtenemos la siguiente salida proporcionándole a CLINGO el programa completo:

% Answer: 1
% vecinos(manuel,julia) vecinos(rocio,julia) vecinos(manuel,marta)
% vecinos(rocio,marta) vecinos(julia,manuel) vecinos(marta,manuel)
% vecinos(julia,rocio) vecinos(marta,rocio)

Obtenemos por tanto la siguiente información nueva:

1. Manuel y Rocío son vecinos de Julia y Marta.
2. Julia y Marta son vecinas de Manuel y Rocío.

**Añadiendo nueva información a la base de conocimientos.**

Sabemos ahora que:

1. Alberto reside en el barrio B.
2. No tenemos conocimiento de que Alberto viva con nadie más.

Modelizamos la información usando de nuevo los predicados **persona/1** y **residencia_B/1**:

```
persona(alberto).
residencia_B(alberto).
```

Si le preguntamos al programa: *¿son Marta y Alberto convivientes?*, su respuesta es *desconocido*, pues no aparecen los literales *convivientes(marta,alberto)* ni *-convivientes(marta,alberto)* en el conjunto de respuesta. Sin embargo, nosotros no tenemos constancia de que Marta y Alberto sean convivientes, y por tanto en nuestro día a día trabajaríamos con la hipótesis de que no lo son a menos que se nos indicase lo contrario.

Modelizamos este conocimiento a través de la *hipótesis del mundo cerrado*: si no se tiene constancia de que dos personas distintas son convivientes, entonces no lo son. Añadimos así la siguiente regla al programa:

```
-convivientes(X,Y) :- not convivientes(X,Y), persona(X),
                      persona(Y), X!=Y.
```

Obtenemos como salida:

% vecinos(julia,alberto) vecinos(marta,alberto)
% vecinos(manuel,alberto) vecinos(rocio,alberto)
% vecinos(alberto,julia) vecinos(manuel,julia)
% vecinos(rocio,julia) vecinos(alberto,marta)
% vecinos(manuel,marta) vecinos(rocio,marta)
% vecinos(alberto,manuel) vecinos(julia,manuel)
% vecinos(marta,manuel) vecinos(alberto,rocio)
% vecinos(julia,rocio) vecinos(marta,rocio)
% -convivientes(julia,alberto) -convivientes(marta,alberto)
% -convivientes(manuel,alberto) -convivientes(rocio,alberto)
% -convivientes(alberto,julia) -convivientes(manuel,julia)
% -convivientes(rocio,julia) -convivientes(alberto,marta)
% -convivientes(manuel,marta) -convivientes(rocio,marta)
% -convivientes(alberto,manuel) -convivientes(julia,manuel)
% -convivientes(marta,manuel) -convivientes(alberto,rocio)
% -convivientes(julia,rocio) -convivientes(marta,rocio)

Hemos obtenido así la siguiente información nueva sobre Alberto:

1. Alberto es vecino de Julia, Rocío, Marta y Manuel.
2. Alberto no convive con Rocío, Julia, Manuel, ni Marta.

Esta regla además no provoca contradicciones, es decir, supongamos ahora que se sabe que Alberto reside en el barrio B y que convive con Marta.

De nuevo, modelizamos esta relación a través del predicado **convivientes/2**:

```
convivientes(alberto,marta).
```

Obtenemos así la salida:

% vecinos(julia,alberto) vecinos(manuel,alberto) vecinos(rocio,alberto)
% vecinos(alberto,julia) vecinos(manuel,julia) vecinos(rocio,julia)
% vecinos(manuel,marta) vecinos(rocio,marta) vecinos(alberto,manuel)
% vecinos(julia,manuel) vecinos(marta,manuel) vecinos(alberto,rocio)
% vecinos(julia,rocio) vecinos(marta,rocio)

% -convivientes(julia,alberto) -convivientes(manuel,alberto)
% -convivientes(rocio,alberto) -convivientes(alberto,julia)
% -convivientes(manuel,julia) -convivientes(rocio,julia)
% -convivientes(manuel,marta) -convivientes(rocio,marta)
% -convivientes(alberto,manuel) -convivientes(julia,manuel)
% -convivientes(marta,manuel) -convivientes(alberto,rocio)
% -convivientes(julia,rocio) -convivientes(marta,rocio)

Aunque sin la regla anterior CLINGO nos devolvía *-convivientes(alberto,marta)* en el conjunto de respuesta, el incluirla no provoca contradicción pues la adición de la nueva regla hace que el literal anterior no aparezca en el conjunto de respuesta al no verificarse el cuerpo de la regla del que procede.

Algunas anotaciones de la regla (suponiendo que Alberto y Marta son convivientes):

1. Cuando CLINGO nos devuelve el conjunto de respuesta de este programa, en él nos aparece el literal *-convivientes(alberto,julia)*. Esto es porque el sentido común no sólo nos dice que la relación "ser convivientes" es simétrica sino que también es transitiva: si X es conviviente con Y e Y es conviviente con Z, entonces X y Z son convivientes, y esta información no aparece reflejada en el programa. La modelizamos a través de la regla:

   ```
   convivientes(X,Z) :- convivientes(X,Y), convivientes(Y,Z),
                        X!=Z.
   ```

   Es necesario poner que X y Z no son la misma persona: en caso contrario, como se verificaría *convivientes(alberto,marta)* y, por la simetría, *convivientes(marta,alberto)*, se obtendría *convivientes(alberto,alberto)*, que no queremos que aparezca en el conjunto de respuesta ya que una persona no convive consigo misma.

2. Si en vez de la regla anterior, hubiésemos puesto la regla:

   ```
   -convivientes(X,Y) :- not convivientes(X,Y), X!=Y.
   ```

   CLINGO nos hubiese devuelto el siguiente mensaje de error (hemos llamado al archivo con las reglas del programa `Ejemplo_Vecinos.lp`):

   % Ejemplo_Vecinos.lp:6:1-51: error: unsafe variables in:
   % (-convivientes(X,Y)):-#inc_base;X!=Y;not convivientes(X,Y).
   % Ejemplo_Vecinos.lp:6:15-16: note: 'X' is unsafe

% Ejemplo_Vecinos.lp:6:17-18: note: 'Y' is unsafe

que nos indica que las variables X e Y no son seguras. Una variable Z se dice *segura* si aparece al menos una vez en el cuerpo de la regla sin estar influenciada por la negación por defecto *not*. En nuestro caso, las variables X e Y aparecen afectadas por *not* y por el operador "!=", que tampoco nos sirve para hacer la regla segura. Por eso, debemos incluir en la regla los literales *persona(X)* y *persona(Y)*.

## 3.3. Modelización de las relaciones familiares

### 3.3.1. Definición recursiva de antepasado

Tenemos el siguiente marco familiar:

1. Antonio y Luisa son los padres de Rosa.
2. Rosa y Marcos tienen un hijo llamado Lucas.
3. Rocío y Lucas son los padres de Julieta.
4. Nerea y Pedro son los padres de Rocío.
5. Suponemos que X no es el padre o la madre de Y si no tenemos constancia de estos hechos.

Modelicemos esta información conocida.

Para clasificar los elementos, usamos el predicado **persona/1**:

```
persona(nerea;pedro;rocio;marcos;antonio;luisa;rosa;lucas;
        julieta).
```

Para representar las relaciones entre ellos, usamos los predicados **padre/2** y **madre/2**:

```
padre(pedro,rocio;antonio,rosa;marcos,lucas;lucas,julieta).
madre(nerea,rocio;rocio,julieta;luisa,rosa;rosa,lucas).
```

Al igual que en la Sección 3.2, usamos la hipótesis del mundo cerrado para modelizar el siguiente conocimiento: si no tenemos constancia de que X sea padre (o madre) de Y, entonces X no es padre (respectivamente madre) de Y. Incluimos por tanto las siguientes dos reglas al programa:

```
-padre(X,Y) :- not padre(X,Y), persona(X), persona(Y).
-madre(X,Y) :- not madre(X,Y), persona(X), persona(Y).
```

*Nota: tenemos que incluir los literales persona(X) y persona(Y) para que la regla sea segura pues tanto X como Y se ven afectadas por la negación por defecto.*

**Concepto: Progenitor.**

Usando los predicados **padre/2** y **madre/2**, definimos también el predicado **progenitor/2**: X es progenitor de Y si es padre o madre de Y. Modelizamos el conocimiento con las reglas:

```
progenitor(X,Y) :- padre(X,Y).
progenitor(X,Y) :- madre(X,Y).
```

Obtenemos con estas reglas la siguiente información:

%progenitor(pedro,rocio) progenitor(antonio,rosa) progenitor(marcos,lucas)
% progenitor(lucas,julieta) progenitor(nerea,rocio) progenitor(rocio,julieta)
% progenitor(luisa,rosa) progenitor(rosa,lucas)

**Concepto: Antepasado.**

Definimos ahora la relación *antepasado* de la siguiente forma recursiva:

- Si X es progenitor de Y, entonces X es antepasado de Y.
- Si X es antepasado de Y e Y es progenitor de Z, entonces X es antepasado de Z.

Implementamos esta información a través de las siguientes reglas:

```
antepasado(X,Y) :- progenitor(X,Y).
antepasado(X,Z) :- progenitor(Y,Z),antepasado(X,Y).
```

Completamos la información utilizando la hipótesis del mundo cerrado: si no tenemos constancia de que X sea antepasado de Y, entonces X no es antepasado de Y.

```
-antepasado(X,Y) :- not antepasado(X,Y),persona(X), persona(Y).
```

Veamos el modelo que nos da CLINGO:

% antepasado(pedro,rocio) antepasado(antonio,rosa) antepasado(marcos,lucas)
% antepasado(lucas,julieta) antepasado(nerea,rocio) antepasado(rocio,julieta)
% antepasado(luisa,rosa) antepasado(rosa,lucas) antepasado(pedro,julieta)
% antepasado(antonio,lucas) antepasado(marcos,julieta)
% antepasado(nerea,julieta) antepasado(luisa,lucas) antepasado(rosa,julieta)
% antepasado(antonio,julieta) antepasado(luisa,julieta)

Obtenemos información nueva, como:

1. Pedro, Marco, Nerea, Rosa, Antonio y Luisa son antepasados de Julieta.
2. Antonio y Luisa son antepasados de Lucas.

### 3.3.2. Definición de hijo único

Poseemos los datos de dos familias:

- La primera está formada por Nerea y Pedro, que tienen una hija, Irene.
- La segunda está formada por Josefa y Rodrigo, que tienen dos hijos, Marcos y Mario.

Veamos cómo podemos implementar la información.

Para clasificar los elementos volvemos a usar el predicado **persona/1** junto con el predicado **genero/2**:

```
persona(nerea;irene;pedro;josefa;rodrigo;marcos;mario).
genero(irene,mujer;nerea,mujer;josefa,mujer).
genero(pedro,hombre;marcos,hombre;mario,hombre;rodrigo,hombre).
```

Para modelizar las relaciones entre ellos, usamos los predicados **padre/2** y **madre/2**:

```
padre(pedro,irene;rodrigo,mario;rodrigo,marcos).
madre(nerea,irene;josefa,marcos;josefa,mario).
```

Si le preguntamos ahora a CLINGO si Nerea es el padre de Marcos, su respuesta es *desconocido*, pues ni *padre(nerea,marcos)* ni *-padre(nerea,marcos)* pertenecen al conjunto de respuesta del programa. Pero sabemos que Nerea no puede ser el padre de Marcos, porque Nerea es una mujer. Para representar este conocimiento implícito, usamos las dos reglas siguientes:

```
-padre(X,Y) :- genero(X,mujer),persona(Y), X!=Y.
-madre(X,Y) :- genero(X,hombre),persona(Y), X!=Y.
```

**Concepto: progenitor.**

De nuevo, implementamos en la base de conocimiento la relación *progenitor*: X es progenitor de Y si X es padre o madre de Y.

```
progenitor(X,Y) :- padre(X,Y).
progenitor(X,Y) :- madre(X,Y).
```

**Concepto: hermanos.**

Supongamos que queremos saber quiénes de los individuos son hermanos. Tenemos que modelizar la relación *hermanos*: X e Y son hermanos si no son la misma persona y tienen los mismos padres. La definición queda representada usando el predicado **hermanos/2**:

```
hermanos(X,Y) :- padre(F,X), padre(F,Y), madre(M,X),madre(M,Y),
                 X != Y.
```

La relación *hermanos* es simétrica y transitiva:

```
hermanos(X,Y) :- hermanos(Y,X).
hermanos(X,Z) :- hermanos(X,Y),hermanos(Y,Z), X != Z.
```

Si le proporcionamos este programa a CLINGO, nos devuelve (nos fijamos sólo en el predicado *hermanos*):

%hermanos(mario,marcos) hermanos(marcos,mario)

Así, CLINGO nos proporciona la información que nos interesaba conocer: Marcos y Mario son hermanos.

**Completitud de la información de la base de conocimiento.**

Utilizamos la hipótesis del mundo cerrado para representar lo siguiente: si no tenemos constancia de que X e Y sean hermanos, entonces X e Y no son hermanos.

```
-hermanos(X,Y) :- not hermanos(X,Y), persona(X), persona(Y).
```

**Conceptos: hijo e hijo único.**

Nos interesa saber quiénes de los individuos son hijos únicos. Para ello, definamos primero la relación *hijo*.

Usando el predicado **progenitor/2**, modelizamos la relación *hijo*: X es hijo de Y si Y es un progenitor de X.

```
hijo(X,Y) :- progenitor(Y,X).
```

Por último, representemos la relación "ser hijo único": X es hijo único si es hijo de alguien y no tiene hermanos (su progenitor no tiene más hijos):

```
hijoUnico(X) :- hijo(X,Y), #false: Z !=X, hijo(Z,Y).
```

De esta manera, proporcionándole a CLINGO todas las reglas anteriores, obtenemos la salida:

%hijoUnico(irene)

Y obtenemos así la información que buscábamos: Irene es hija única.

## 3.4. Modelización de un conjunto de medios de transporte

En este último ejemplo, vamos a ver cómo modelizar una base jerárquica de conocimiento: los medios de transporte.

Disponemos de la siguiente información:

1. Los vehículos se dividen en marítimos y no marítimos.
2. Los submarinos, los veleros y las lanchas son vehículos marítimos.
3. Los autobuses y las caravanas son vehículos no marítimos.
4. Desvaste es una caravana concreta.
5. Ultramar es un vehículo marítimo concreto.

Estudiemos cómo modelizar este conocimiento jerárquico del que disponemos.

Usaremos el predicado **clase/1** para enumerar las clases que constituyen la jerarquía:

```
clase(vehiculo;mar;no_mar;submarino;velero;lancha;
      autobus;caravana).
```

Para describir la jerarquía entre ellas, usamos el predicado **subclase_de/2**, que representa la contención inmediata o directa entre clases:

```
subclase_de(mar,vehiculo).
subclase_de(no_mar,vehiculo).
subclase_de(submarino,mar).
subclase_de(velero,mar).
subclase_de(lancha,mar).
subclase_de(autobus,no_mar).
subclase_de(caravana,no_mar).
```

**Concepto: subclase.**

Vamos a modelizar la clausura transitiva de la relación **subclase_de/2**:

1. Si C1 es subclase directa de C2, entonces C1 es subclase de C2.
2. Si C1 es subclase directa de C2 y C2 es subclase de C3, entonces C1 es subclase de C3.

Para la implementación de la relación, se usa el predicado **subclase/2**:

```
subclase(C1,C2) :- subclase_de(C1,C2).
subclase(C1,C3) :- subclase_de(C1,C2), subclase(C2,C3).
```

*Nota: análogamente se puede modelizar la clausura transitiva de cualquier otra relación binaria.*

Completemos la información sobre el predicado **subclase/2**: si no tenemos constancia de que la clase C2 sea subclase de la clase C1, entonces supondremos que C2 no es subclase de C1. Al igual que en las secciones anteriores, representamos este conocimiento con la hipótesis del mundo cerrado:

```
-subclase(C1,C2) :- clase(C1), clase(C2), not subclase(C1,C2).
```

De esta manera, obtenemos el siguiente conjunto de respuesta usando CLINGO:

% subclase(mar,vehiculo) subclase(no_mar,vehiculo) subclase(submarino,mar)
% subclase(velero,mar) subclase(lancha,mar) subclase(autobus,no_mar)
% subclase(caravana,no_mar) subclase(submarino,vehiculo)
% subclase(velero,vehiculo) subclase(lancha,vehiculo)
% subclase(autobus,vehiculo) subclase(caravana,vehiculo)

Obtenemos la siguiente información que no había sido proporcionada al agente acerca de las subclases de la jerarquía: lancha, velero, submarino, autobús y caravana son subclases de vehículo.

### Concepto: pertenencia directa e indirecta a una clase.

Modelicemos la información que nos queda por representar: Desvaste es una caravana concreta y Ultramar es un vehículo marítimo concreto. Para la representación, usamos el predicado **elemento/1**:

```
elemento(desvaste).
elemento(ultramar).
```

Para representar la pertenencia "directa" de un elemento a una clase, usamos el predicado **es_un/2**:

```
es_un(desvaste,caravana).
es_un(ultramar,mar).
```

Definamos el concepto de pertenencia a una clase: X pertenece a una clase C si es un elemento de dicha clase ("pertenencia directa") o pertenece de manera directa a una subclase de la clase C.

Modelizamos la información a través de las dos reglas siguientes:

```
pertenece(X,C) :- es_un(X,C).
pertenece(X,C2) :- es_un(X,C1), subclase(C1,C2).
```

*Nota: esta definición es independiente del contexto particular y puede usarse para modelizar la noción de pertenencia.*

De este modo, CLINGO nos devuelve todas las clases a las que pertenece Desvaste y Ultramar:

% pertenece(desvaste,caravana) pertenece(ultramar,mar)
% pertenece(desvaste,no_mar) pertenece(desvaste,vehiculo)
% pertenece(ultramar,vehiculo)

Disponemos así de información nueva acerca de los elementos Desvaste y Ultramar:

1. Desvaste y Ultramar son vehículos.
2. Desvaste es un vehículo no marítimo.

**Concepto: clases hermanas.**

Si le preguntamos a CLINGO si Desvaste es un autobús o si Ultramar es un vehículo no marítimo, este nos responde *desconocido* pues los literales *pertenece(desvaste,autobus)* y *-pertenece(desvaste,autobus)* no aparecen en el conjunto de respuesta, y del mismo modo tampoco aparecen los literales *pertenece(ultramar,no_mar)* y *-pertenece(ultramar,no_mar)*. Sin embargo, sabemos que la respuesta debería ser *no*, pues Desvaste ya pertenece a la clase caravana (luego no puede ser un autobús) y Ultramar a la clase marítima (luego no puede pertenecer a la clase no marítima).

Para representar este conocimiento, definimos primero *clases hermanas*: X e Y son clases hermanas si son distintas y son subclases directas de la misma clase. Para implementar la relación, usamos el predicado **hermanas/2**:

```
hermanas(C1,C2) :- C1 != C2, subclase_de(C1,C3),
                   subclase_de(C2,C3).
```

A través de la siguiente regla representamos que un elemento que pertenece a una clase C1 no puede pertencer a ninguna clase C2 que sea hermana de C1:

```
-pertenece(X,C2) :- pertenece(X,C1), hermanas(C1,C2).
```

Finalmente, CLINGO nos devuelve el conjunto de respuesta:

% hermanas(no_mar,mar) hermanas(mar,no_mar)
% hermanas(velero,submarino) hermanas(lancha,submarino)
% hermanas(submarino,velero) hermanas(lancha,velero)
% hermanas(submarino,lancha) hermanas(velero,lancha)
% hermanas(caravana,autobus) hermanas(autobus,caravana)
% -pertenece(desvaste,autobus) -pertenece(ultramar,no_mar)
% -pertenece(desvaste,mar)

de manera que se conoce la siguiente información nueva sobre los elementos:

1. Desvaste no es un autobús. Esto es gracias a que las clases autobús y caravana son hermanas, y Desvaste es una caravana.

2. Desvaste no es un vehículo marítimo. Esto se debe a que Desvaste pertenece a la clase no marítimo y las clases no marítimo y marítimo son hermanas. Ultramar no es un vehículo no marítimo por un razonamiento análogo al que acabamos de exponer.

# Capítulo 4

# Representación por defecto

En este capítulo, vamos a tratar cómo se representan los *defectos*, esto es, información en la que se incluye expresiones como *normalmente, generalmente*, etc., y cómo representar las excepciones que un defecto puede tener.

## 4.1. Representación por defecto y tipos de excepciones

Como ya hemos mencionado en otras ocasiones, nuestro objetivo es modelizar contextos reales. En estos contextos, podemos encontrar situaciones marcadas por expresiones como *normalmente, generalmente,* etc. Esto es lo que se conoce como *defecto*. De esta manera, podemos tener situaciones en las que el defecto no se cumple o no sepamos si se cumple. Estas situaciones se denominan *excepciones*. Las conclusiones que saquemos de los defectos serán provisionales, pudiendo ser refutadas posteriormente.

Veamos esto a través de un ejemplo. Consideremos la base de conocimiento de un barrio B que expusimos en la sección anterior. Se dispone de la siguiente información:

1. Marta, Manuel y Rocío viven en el barrio B.
2. Manuel y Rocío viven juntos.
3. Defecto 1: normalmente, los vecinos del barrio B son amigos.

4. Marta y Manuel no son amigos.

Modelicemos esta información retomando las relaciones que se definieron en la base de conocimiento de un barrio.

Usamos en primer lugar el predicado **persona/1** para clasificar los elementos:

```
persona(marta;manuel;rocio).
```

Representamos que los sujetos viven en el barrio B usando el predicado **residencia_B/1**:

```
residencia_B(marta;manuel;rocio).
```

y usamos el predicado **convivientes/2** para representar que Manuel y Rocío viven juntos:

```
convivientes(manuel,rocio).
```

La relación "ser convivientes" es simétrica y transitiva:

```
convivientes(X,Y) :- convivientes(Y,X).
convivientes(X,Z) :- convivientes(X,Y), convivientes(Y,Z), X!=Z.
```

Por último, vimos el concepto "ser vecinos": X e Y son vecinos si viven en el barrio B y no consta que sean convivientes.

```
vecinos(X,Y) :- residencia_B(X), residencia_B(Y),

                not convivientes(X,Y), X != Y.
```

### Intento de representación del defecto 1.

Representemos ahora el defecto 1: *normalmente*, los vecinos del barrio B son amigos. Usaremos el predicado **amigos/2** para su representación. Inicialmente, podemos pensar en representarlo como: X e Y son amigos si son vecinos. Para ello, se introduce la siguiente regla al programa:

```
amigos(X,Y) :- vecinos(X,Y).
```

Si incluimos ahora que Manuel y Marta no son amigos a través de la regla:

```
-amigos(marta,manuel;manuel,marta).
```

el programa resulta inconsistente, pues el conjunto de respuesta no puede contener contradicciones (no puede contener a los literales *amigos(marta,manuel), amigos(manuel,marta)*, que derivan de la primera regla, a la vez que a los literales *-amigos(marta,manuel), -amigos(manuel,marta)*).

Necesitamos definir por tanto los defectos y sus excepciones de una forma distinta.

En ASP, se utiliza la regla:

`p(X) ← c(X), not ab(d(X)), not ¬p(X).`

para representar el defecto d: "normalmente, los elementos de c tienen la propiedad p". El literal extendido *not ab(d(X))* se usa para representar que no se tiene constancia de que el defecto d no se pueda aplicar a X y el literal *not ¬p(X)* para representar que no se tiene constancia de que X no cumpla la propiedad p (es decir, puede que p(X) se verifique). Es decir, podemos entender la regla como "si X es de la clase c, no se tiene constancia de que no verifique p y tampoco se tiene constancia de que sea una excepción, entonces X verifica p".

Distinguimos dos tipos de excepciones: las *excepciones fuertes* y las *excepciones débiles*.

Las *excepciones débiles* e(X) de un defecto d se representan mediante el *axioma de cancelación*:

`ab(d(X)) ← not ¬e(X).`

que representa que si no se tiene constancia de que X no sea una excepción débil de la regla, entonces el defecto d no puede aplicarse a X. Las excepciones débiles hacen que el agente que razona sobre el contexto no sea capaz de llegar a una conclusión sobre X (desconoce si X verifica o no la propiedad p).

Por otro lado, las *excepciones fuertes* van a refutar el defecto d: el agente obtiene una conclusión opuesta a la que deriva del defecto d. Para representar las excepciones fuertes e(X), se añade al axioma de cancelación anterior la regla:

`¬p(X) :- e(X).`

la cual hace que el defecto d no pueda verificarse. Además, el agente concluye que X no cumple la propiedad p.

### Representación del defecto 1.

Sigamos con el ejemplo. Para representar el defecto "normalmente, los vecinos son amigos", usamos la regla:

```
amigos(X,Y) :- vecinos(X,Y), not ab(d_amigos(X,Y)),
               not -amigos(X,Y).
```

La relación "ser amigos" es simétrica:

```
amigos(X,Y) :- amigos(Y,X).
```

Además, si X no es amigo de Y, Y tampoco es amigo de X:

```
-amigos(X,Y) :- -amigos(Y,X).
```

La información que disponemos acerca de que Marta y Manuel no son amigos es una excepción fuerte, y se representa a través de la regla:

```
-amigos(marta,manuel).
```

De esta manera, el programa ya no es inconsistente y se obtiene la siguiente salida:

% -amigos(marta,manuel) -amigos(manuel,marta) amigos(rocio,marta)
% amigos(marta,rocio)

Obteniendo así información acerca de quiénes de los sujetos son amigos: Marta y Rocío son amigas.

### Modelización de nueva información.

Se dispone ahora de la siguiente información:

1. Se sabe si algunas de las personas del barrio se conocen entre ellas.
2. Se sabe que dos personas que no se conocen no pueden ser amigas.

Para modelizar esta información, se usa el predicado **conocidos/2**, que representa que X conoce a Y e Y conoce a X.

La relación "ser conocidos" es simétrica: si se verifica que X e Y son conocidos, tanto X conoce a Y como Y conoce a X, luego también se verifica que Y y X son conocidos. Representamos el carácter simétrico de la información a través de:

```
conocidos(X,Y) :- conocidos(Y,X).
```

Por motivos similares, si no se verifica que X e Y sean conocidos, tampoco puede verificarse que Y y X sean conocidos, lo que representamos con la regla:

```
-conocidos(X,Y) :- -conocidos(Y,X).
```

Representemos ahora la excepción fuerte: si X e Y no se conocen, entonces no pueden ser amigos.

```
-amigos(X,Y) :- -conocidos(X,Y).
```

De este modo, si no se tiene constancia de que X e Y se conozcan, entonces no debe aplicarse el defecto 1 ya que X e Y pueden ser una excepción. Representamos esta excepción débil a través de la regla:

```
ab(d_amigos(X,Y)) :- not conocidos(X,Y), persona(X), persona(Y).
```

Veamos entonces qué nuevo conocimiento obtenemos según la información sobre el predicado **conocidos/2** de la que se disponga:

**Ejemplo 1**.

Se posee la siguiente información acerca de **conocidos/2**:

- Marta y Manuel se conocen.
- Marta y Manuel no son amigos.

Representamos esta información con las reglas:

```
conocidos(manuel, marta).
-amigos(marta, manuel).
```

Obtenemos la salida (fijándonos en el predicado **amigos/2** y **-amigos/2**):

% -amigos(marta,manuel) -amigos(manuel,marta)

Como no tenemos constancia de que Marta y Rocío se conozcan, no obtenemos información sobre si son o no amigas.

**Ejemplo 2.**

Se posee la siguiente información:

- Marta y Manuel se conocen.
- Marta y Manuel no son amigos.
- Marta y Rocío se conocen.

Añadimos a las dos reglas anteriores la regla:

```
conocidos(marta,rocio).
```

Obtenemos así:

% -amigos(marta,manuel) -amigos(manuel,marta) amigos(rocio,marta)
% amigos(marta,rocio)

Luego disponemos de nueva información: Marta y Rocío son amigas.

**Ejemplo 3.**

Se posee la siguiente información:

- Marta y Manuel se conocen.
- Marta y Manuel no son amigos.
- Marta y Rocío no se conocen.

CLINGO nos proporciona el conjunto de respuesta:

% -amigos(marta,manuel) -amigos(marta,rocio) -amigos(rocio,marta)
% -amigos(manuel,marta)

Y conseguimos de este modo nueva información: Marta y Rocío no son amigas.

### Modelización de una excepción fuerte usando el axioma de cancelación.

Aunque antes se haya modelizado una excepción fuerte sin el axioma de cancelación (pues no era necesario), en otras ocasiones debemos

implementarlo también en el programa. Veamos un ejemplo.

Supongamos que poseemos la siguiente información:

1. Marta, Manuel y Rocío viven en el barrio B.
2. Manuel y Rocío viven juntos.
3. Defecto 1: normalmente, los vecinos el barrio B son amigos.
4. Las personas de la casa c1 y las personas de la casa c2 no son amigas.
5. Manuel vive en la casa c1.
6. Una persona no puede vivir en más de una casa.
7. Se conoce información sobre quiénes de los sujetos, aparte de Manuel, viven en c1 o c2.

Representamos las dos casas usando el predicado **casa/1**:

```
casa(c1).
casa(c2).
```

Representamos que Manuel vive en la casa c1 con el predicado **vivirEn/2**:

```
vivirEn(manuel,c1).
```

Una propiedad acerca de la relación "vivir en" es que si una persona X vive en una casa C y es conviviente con otra persona Y, entonces Y vive en C:

```
vivirEn(Y,C) :- vivirEn(X,C), convivientes(X,Y).
```

Usando la siguiente regla, modelizamos que una persona no puede vivir en más de una casa:

```
-vivirEn(X,C) :- vivirEn(X,C1), C!=C1, casa(C).
```

Representemos ahora la excepción fuerte: si una persona vive en c1 y otra persona vive en c2, entonces no son amigas. Se usa para ello la regla siguiente:

```
-amigos(X,Y) :- vivirEn(X,c1),vivirEn(Y,c2).
```

Necesitamos acompañar la regla con el axioma de cancelación para representar que si no tenemos constancia de que ni X ni Y viven en c1 o c2,

entonces puede que X e Y no sean amigos. Por tanto, usamos las siguientes dos reglas:

```
ab(d_amigos(X,Y)) :- not -vivirEn(X,c1), not -vivirEn(Y,c2),
                     persona(X), persona(Y).
ab(d_amigos(X,Y)) :- not -vivirEn(Y,c1), not -vivirEn(X,c2),
                     persona(X), persona(Y).
```

Veamos como antes qué conocimiento nuevo podemos obtener según la información que poseamos del predicado **vivirEn/2**.

**Ejemplo 1.**

Se dispone de la siguiente información:

- Marta, Manuel y Rocío se conocen entre ellos.
- Marta no vive en c2.

Representamos que Marta, Manuel y Rocío se conocen entre ellos:

```
conocidos(marta,manuel;marta,rocio;manuel,rocio).
```

Representamos ahora que Marta no vive en c2:

```
-vivirEn(marta,c2).
```

En este caso, obtenemos la salida:

% amigos(manuel,marta) amigos(rocio,marta) amigos(marta,manuel)
% amigos(marta,rocio)

Luego hemos obtenido la siguiente información:

1. Marta y Rocío son amigas.
2. Marta y Manuel son amigos.

**Ejemplo 2.**

Se dispone de la siguiente información:

- Marta, Manuel y Rocío se conocen entre ellos.
- Marta vive en c2.

Representamos el conocimiento a través de las reglas:

```
conocidos(marta,manuel;marta,rocio;manuel,rocio).
vivirEn(marta,c2).
```

Obtenemos la salida:

% -amigos(manuel,marta) -amigos(rocio,marta) -amigos(marta,manuel)
% -amigos(marta,rocio)

Luego obtenemos la siguiente información:

1. Marta y Manuel no son amigos.
2. Rocío y Marta no son amigas.

**Ejemplo 3.**

No disponemos información acerca de dónde vive Marta.

En este caso, no obtenemos en el conjunto de respuesta nada acerca de si Marta es amiga o no de Manuel y Rocío. Esto es porque se aplica el axioma de cancelación anterior, que añadimos a la excepción fuerte.

Si no lo hubiésemos considerado, el defecto se aplicaría y el agente consideraría que son amigos entre ellos.

**Caso especial: defectos con información completa.**

Sea el defecto d: "normalmente, los elementos de c verifican la propiedad p" y supongamos que se tiene un conjunto $e$ de excepciones. Si la información que se tiene sobre $e$ es completa, entonces el axioma de cancelación en las excepciones débiles se reduce a:

```
ab(d(X)) :- e(X).
```

y en las excepciones fuertes, dicho axioma puede ser omitido.

En el ejemplo descrito anteriormente, si hubiésemos tenido una lista con las personas que viven en la casa $c_1$ y con las personas que viven en la casa $c_2$, la excepción fuerte se hubiese representado únicamente por la regla:

```
-amigos(X,Y) :- vivirEn(X,c1),vivirEn(Y,c2).
```

pues una persona de la que no se conoce dónde vive deja de ser una posible excepción ya que, al ser la información sobre las personas que viven en $c_1$ o $c_2$ completa, si viviese en una de estas casas, el agente lo sabría.

## 4.2. Modelización de la información con valores nulos

Los defectos pueden usarse también para modelizar información en la que aparecen *valores nulos*, esto es, constantes que indican que el valor de una variable o función es desconocido.

Supongamos que tenemos una empresa con varios departamentos, y disponemos de la siguiente información:

1. Nuria es la encargada del departamento $D_1$.
2. Luis es el encargado del departamento $D_2$.
3. En los departamentos $D_3$ y $D_4$ se va a contratar a una persona como encargado entre las que lo soliciten.
4. Defecto 1: normalmente, si una persona no está en la lista de encargados junto a un departamento, no se encarga de ese departamento.
5. Los solicitantes de los puestos vacantes son Nuria y Marcos.

La lista de los encargados con sus respectivos departamentos es la siguiente:

| Persona | Departamento |
|---|---|
| Nuria | $D_1$ |
| Luis | $D_2$ |
| Vacante | $D_3$ |
| Vacante | $D_4$ |

En este ejemplo, "Vacante" es un valor nulo que nos indica que las personas encargadas de los departamentos $D_3$ y $D_4$ están por determinar.

Veamos cómo modelizar esta información.

En primer lugar, representamos los elementos que intervienen en el ejemplo con los predicados **persona/1** y **departamento/1**:

```
persona(nuria;luis;marcos).
departamento(d1;d2;d3;d4).
```

Representamos la información de la lista de encargados con el predicado **encargado/2**:

```
encargado(nuria,d1;luis,d2;vacante,d3;vacante,d4).
```

y modelizamos también con el predicado **solicitante/1** los sujetos que han solicitado los puestos vacantes:

```
solicitante(nuria;marcos).
```

Representamos el defecto 1 a través de la regla:

```
-encargado(X,D) :- persona(X), departamento(D),
                   not ab(d_encargado(X,D)), not encargado(X,D).
```

que se lee como "si no tenemos constancia de que X pueda ser una excepción del defecto y tampoco tenemos constancia de que X sea la persona encargada del departamento D, es decir, no tenemos constancia de que X sea una excepción, entonces X no es la persona encargada del departamento D".

Sin embargo, los encargados de los departamentos $D_3$ y $D_4$ están por determinar: se contratará a algún solicitante aunque sus nombres no aparezcan en la lista junto a estos departamentos. Esta información por tanto es una excepción débil del defecto 1 y se modeliza con la regla:

```
ab(d_encargado(X,D)) :- solicitante(X), encargado(vacante,D).
```

Esta regla representa el siguiente conocimiento: "si X es un solicitante, entonces puede ser un encargado de los departamentos D con puestos vacantes aunque su nombre no aparezca en la lista junto a ellos".

De esta manera, obtenemos la siguiente salida:

% encargado(nuria,d1) encargado(luis,d2) encargado(vacante,d3)
% encargado(vacante,d4) -encargado(nuria,d2) -encargado(luis,d1)
% -encargado(luis,d3) -encargado(luis,d4) -encargado(marcos,d1)
% -encargado(marcos,d2)

Así, se dispone de la siguiente información nueva:

- Nuria no se encarga del departamento $D_2$.
- Luis no se encarga de los departamentos $D_1$, $D_3$ y $D_4$.
- Marcos no se encarga de los departamentos $D_1$ y $D_2$.

No obtenemos información sobre si Nuria y Marcos son o no encargados de los departamentos $D_3$ y $D_4$ ya que son los solicitantes, figuran como posibles excepciones.

También podríamos haber representado la información del siguiente modo:

| Persona | Departamento |
|---|---|
| Nuria | $D_1$ |
| Luis | $D_2$ |
| Vacante | $D_3$ |
| Vacante | $D_4$ |
| {Nuria, Marcos} | $D_3$ |
| {Nuria, Marcos} | $D_4$ |

donde ahora el valor nulo "Vacante" nos indica que los encargados de los departamentos $D_3$ y $D_4$ están por contratar y el valor nulo "{Nuria, Marcos}" nos indica quiénes son los posibles encargados de los susodichos departamentos. Para representar esta información, sustituimos las dos últimas reglas del programa:

```
solicitante(nuria;marcos).
ab(d_encargado(X,D)) :- solicitante(X), encargado(vacante,D).
```

por las reglas:

```
encargado(nuria,d3) | encargado(marcos,d3).
encargado(nuria,d4) | encargado(marcos,d4).
```

que representan, a través de la disyunción epistémica, que Nuria y Marcos podrían ser los encargados de los departamentos $D_3$ y $D_4$.

Se obtienen cuatro posibles soluciones:

```
%Answer: 1
% encargado(nuria,d1) encargado(luis,d2) encargado(vacante,d3)
% encargado(vacante,d4) -encargado(nuria,d2) -encargado(luis,d1)
% -encargado(luis,d3) -encargado(luis,d4) -encargado(marcos,d1)
% -encargado(marcos,d2) encargado(nuria,d3) encargado(marcos,d4)

%Answer: 2
% encargado(nuria,d1) encargado(luis,d2) encargado(vacante,d3)
% encargado(vacante,d4) -encargado(nuria,d2) -encargado(luis,d1)
% -encargado(luis,d3) -encargado(luis,d4) -encargado(marcos,d1)
% -encargado(marcos,d2) encargado(nuria,d3) encargado(nuria,d4)

%Answer: 3
% encargado(nuria,d1) encargado(luis,d2) encargado(vacante,d3)
% encargado(vacante,d4) -encargado(nuria,d2) -encargado(luis,d1)
% -encargado(luis,d3) -encargado(luis,d4) -encargado(marcos,d1)
% -encargado(marcos,d2) encargado(marcos,d3) encargado(marcos,d4)

%Answer: 4
% encargado(nuria,d1) encargado(luis,d2) encargado(vacante,d3)
% encargado(vacante,d4) -encargado(nuria,d2) -encargado(luis,d1)
% -encargado(luis,d3) -encargado(luis,d4) -encargado(marcos,d1)
% -encargado(marcos,d2) encargado(marcos,d3) encargado(nuria,d4)
```

Todos los conjuntos de respuesta nos proporcionan la siguiente información:

1. Ni Luis ni Marcos son los encargados del departamento $D_1$.
2. Ni Nuria ni Marcos son los encargados del departamento $D_2$.
3. Luis no se encarga de los departamentos $D_3$ y $D_4$.

En la primera solución, obtenemos:

- Nuria es la encargada del departamento $D_3$.
- Marcos es el encargado del departamento $D_4$.

En la segunda, obtenemos:

- Nuria es la encargada de los departamentos $D_3$ y $D_4$.

En la tercera:

- Marcos se encarga de los departamentos $D_3$ y $D_4$.

Y por último, en la cuarta:

- Nuria es la encargada del departamento $D_4$.
- Marcos es el encargado del departamento $D_3$.

## 4.3. Prioridad entre defectos

Como hemos visto, en los contextos reales nos aparecen de manera natural situaciones que se rigen por defectos. En este apartado, vamos a estudiar cómo modelizar la prioridad entre ellos.

Con este fin, retomamos la base de conocimiento de una familia que ya expusimos con anterioridad.

Supongamos que se dispone de los siguientes datos:

1. Ricardo y Susana tienen tres hijos, Pedro, Andrea y Lola.
2. Ruben y Pilar sólo tienen una hija, Cristina.
3. Existen dos tipos de descuentos en el trasporte, d1 y d2.
4. Defecto 1: normalmente, si una persona pertenece a una familia numerosa, se le aplica el descuento d1 en el transporte.
5. Defecto 2: normalmente, a las personas jóvenes se les hace un descuento d2 en el trasporte.
6. Restricción: a una persona no se le puede aplicar los dos descuentos.
7. Es preferible d1 a d2.

8. Si una persona no es joven, no se le puede aplicar ninguno de los dos descuentos.

9. Información sobre si algunas personas son o no jóvenes.

Comencemos modelizando la información disponible.

Usamos el predicado **persona/1** para clasificar los elementos:

```
persona(susana;andrea;ricardo;pedro;cristina;pilar;ruben;lola).
```

Representamos la relación de parentesco entre los sujetos usando los predicados **padre/2** y **madre/2**:

```
padre(ricardo, andrea; ricardo, pedro;ricardo,lola;ruben,cristina).
madre(susana,andrea; susana, pedro;susana,lola;pilar,cristina).
```

Usando el predicado **descuentos/1**, modelizamos la existencia de los descuentos d1 y d2:

```
descuentos(d1;d2).
```

Ahora, vamos a retomar el contexto familiar donde vamos a trabajar.

Definimos el concepto "hermanos": X e Y son hermanos si poseen la misma madre y el mismo padre.

```
hermanos(X,Y) :- padre(F,X), padre(F,Y), madre(M,X), madre(M,Y),

                 X != Y.
```

El predicado **hermanos/2** es simétrico y transitivo:

```
hermanos(X,Y) :- hermanos(Y,X).
hermanos(X,Z) :- hermanos(X,Y),hermanos(Y,Z), X != Z.
```

Completamos la información sobre la relación "hermanos" usando la hipótesis del mundo cerrado:

```
-hermanos(X,Y) :- not hermanos(X,Y), persona(X), persona(Y).
```

Modelizamos ahora la relación "progenitor": X es progenitor de Y si es padre o madre de Y.

```
progenitor(X,Y) :- padre(X,Y).
progenitor(X,Y) :- madre(X,Y).
```

Modelizamos "hijo": X es hijo de Y si Y es un progenitor de X.

```
hijo(X,Y) :- progenitor(Y,X).
```

Como vimos, usando el predicado **hijo/2** se define el predicado **hijoUnico/1**:

```
hijoUnico(X) :- hijo(X,Y), #false: Z !=X, hijo(Z,Y).
```

### Definición de familia numerosa y modelización de los defectos.

X pertenece a una familia numerosa si es joven, posee al menos dos hermanos y estos son jóvenes:

```
familiaNumerosa(X) :- Z != Y , hermanos(X,Y), hermanos(X,Z),
                      joven(X), joven(Y), joven(Z).
```

Representemos ahora el defecto 1: normalmente, si una persona pertenece a una familia numerosa, se le aplica el descuendo d1 en el transporte.

```
descuento(d1,X) :- familiaNumerosa(X), not ab(d1(X)),
                   not -descuento(d1,X).
```

Para implementar el defecto 2, "normalmente, a las personas jóvenes se les hace un descuento d2 en el trasporte", se usa el predicado **joven/1**, que nos indica si X es joven o no:

```
descuento(d2,X) :- joven(X), not ab(d2(X)), not -descuento(d2,X).
```

Por último, modelicemos que a una persona no se le puede aplicar más de un descuento:

```
-descuento(D,X) :- descuento(C,X), C != D, descuentos(D),
                   descuentos(C).
```

### Modelización de la prioridad entre defectos.

Tenemos que representar que el descuento d1 es preferible al descuento d2, es decir, que si a una persona se le puede aplicar el descuento d1 (pertenece a

una familia numerosa), entonces no se le aplica el descuento d2. La prioridad entre defectos es por tanto una excepción fuerte y se modeliza a través de la regla:

```
-descuento(d2,X) :- familiaNumerosa(X), persona(X).
```

Como la información acerca de quién pertenece a una familia numerosa no es completa, si no tenemos constancia de que X pertenezca a una familia numerosa, no se le puede aplicar el descuendo d1:

```
ab(d1(X)) :- not familiaNumerosa(X), persona(X).
```

Por el contrario, la información acerca de cuáles de los sujetos son jóvenes es completa, y usamos la hipótesis del mundo cerrado para representarlo:

```
-joven(X) :- not joven(X), persona(X).
```

Por último, representamos que si una persona no es joven, no tiene derecho a ningún descuento:

```
-descuento(D,X) :- -joven(X), descuentos(D), persona(X).
```

Ahora, estudiemos qué nuevos datos podemos obtener según la información de la que se disponga sobre el predicado **joven/1**.

**Ejemplo 1.**

Andrea, Pedro, Lola y Cristina son jóvenes.

Se obtiene la siguiente solución:

% descuento(d1,pedro) descuento(d1,lola) descuento(d1,andrea)
% descuento(d2,cristina)

Y obtenemos los siguientes nuevos datos:

- A Pedro, Lola y Andrea se le aplica el descuento d1. Esto es coherente, pues aunque son jóvenes y pueden obtener d2, tienen derecho también a d1, que es preferible a d2.
- A Cristina se le aplica el descuento d2.

**Ejemplo 2.**

Se conoce que Andrea, Lola y Cristina son jóvenes.

Se obtiene en este caso la solución:

%descuento(d2,andrea) descuento(d2,lola) descuento(d2,cristina)

Así, la información nueva que se ha obtenido es que a Andrea, Lola y a Cristina se les aplica el descuento d2. Esto ocurre porque como no tenemos constancia de que Pedro sea joven, se obtiene que no es joven, y por lo tanto ningún sujeto tiene derecho a obtener el descuento d1.

## 4.4. Defectos en bases de conocimiento jerárquicas

Por último, retomemos la base de conocimiento de los medios de transporte para ver cómo podemos representar defectos en bases de conocimiento jerárquicas.

Como ya vimos, los vehículos se clasifican en marítimos y no marítimos. Los submarinos, los veleros y las lanchas son vehículos marítimos mientras que los autobuses y las caravanas son vehículos no marítimos. También disponíamos de información sobre Desvaste y Ultramar, que son una caravana y un vehículo marítimo concreto respectivamente.

Veamos la nueva información de la que disponemos:

1. Defecto 1: en general, las caravanas son blancas.
2. Defecto 2: los vehículos no marítimos son habitualmente de color negro.
3. Defecto 3: generalmente, los autobuses son de color naranja.
4. Defecto 4: normalmente, los autobuses turísticos son de color azul.
5. Defecto 5: en general, los vehículos marítimos son de color blanco.
6. Un elemento sólo puede ser de un color.

Comenzamos por tanto retomando la modelización de la base de conocimiento de los medios de transporte.

Representamos las distintas clases de la jerarquía usando el predicado **clase/1**:

```
clase(vehiculo;mar;no_mar;submarino;velero;lancha;autobus;
      caravana;turistico).
```

Usamos el predicado **subclase_de/2** para representar la contención inmediata de una clase en otra:

```
subclase_de(mar,vehiculo;no_mar,vehiculo;submarino,mar;velero,mar;
      lancha,mar;autobus,no_mar;caravana,no_mar;turistico,autobus).
```

Definimos **subclase/2** como la clausura transitiva del predicado **subclase_de/2**:

```
subclase(C1,C2) :- subclase_de(C1,C2).
subclase(C1,C3) :- subclase_de(C1,C2), subclase(C2,C3).
```

Aplicamos la hipótesis del mundo cerrado al predicado **subclase/2**:

```
-subclase(C1,C2) :- clase(C1), clase(C2), not subclase(C1,C2).
```

Representamos que Desvaste es una caravana y que Ultramar es un vehículo marítimo con el predicado **elemento/1**:

```
elemento(desvaste).
elemento(ultramar).
```

Para representar la pertenencia directa de un elemento a una clase, usamos el predicado **es_un**:

```
es_un(desvaste,caravana).
es_un(ultramar,mar).
```

Retomamos la definición del predicado **pertenencia/2**:

```
pertenece(X,C) :- es_un(X,C).
pertenece(X,C2) :- es_un(X,C1), subclase(C1,C2).
```

Definimos "clases hermanas": X e Y son hermanas si son subclases directas de la misma clase.

```
hermanas(C1,C2) :- C1 != C2, subclase_de(C1,C3), subclase_de(C2,C3).
```

Si un elemento pertenece a un clase C1, entonces el elemento no puede pertenecer a ninguna clase C2 que sea hermana de C1.

```
-pertenece(X,C2) :- pertenece(X,C1), hermanas(C1,C2).
```

**Modelización de defectos y excepciones.**

Representamos los colores a través del predicado **color/1**:

```
color(naranja;blanco;negro;azul).
```

Antes de empezar con la representación de los defectos, modelizamos la siguiente información conocida: un elemento sólo puede ser de un color.

```
-deColor(X,C) :- deColor(X,C1), C1 !=C, color(C), color(C1).
```

Representamos el defecto 1, "en general, las caravanas son blancas", como sigue:

```
deColor(X,blanco) :- pertenece(X,caravana), not ab(d1(X)),
                     not -deColor(X,blanco).
```

Representamos el defecto 2, "los vehículos no marítimos son habitualmente de color negro", a través de la regla:

```
deColor(X,negro) :- pertenece(X,no_mar), not ab(d2(X)),
                    not -deColor(X,negro).
```

Para modelizar el defecto 3: los autobuses normalmente son naranjas, se utiliza:

```
deColor(X, naranja) :- pertenece(X,autobus), not ab(d3(X)),
                       not -deColor(X,naranja).
```

Representamos el defecto 4, "normalmente, los autobuses turísticos son de color azul", con la regla:

```
deColor(X,azul) :- pertenece(X,turistico), not ab(d4(X)),
                   not -deColor(X,azul).
```

Y por último, se modeliza el defecto 5, "en general, los vehículos marítimos son blancos", con la regla:

```
deColor(X,blanco) :- pertenece(X,mar), not ab(d5(X)),
                     not -deColor(X,blanco).
```

La información acerca de si X es una caravana no es completa: si no tenemos constancia de que X, que es un vehículo no marítimo, no es una caravana, puede ser una excepción del defecto 2 (puede que no sea de color negro). Esto se representa a través de una excepción débil:

```
ab(d2(X)) :- not -pertenece(X,caravana), elemento(X).
```

De la misma manera, la información sobre si X es un autobús o un autobús turístico no es completa. Por este motivo, si no tenemos constancia de que X no sea alguno de estos vehículos, puede que X sea una excepción de los defectos 2 y 3 respectivamente. Representamos de nuevo la información como excepciones débiles:

```
ab(d2(X)) :- not -pertenece(X,autobus), elemento(X).
ab(d3(X)) :- not -pertenece(X,turistico), elemento(X).
```

Veamos entonces qué información nos proporciona CLINGO sobre los colores de la caravana Desvaste y el vehículo marítimo Ultramar. Obtenemos la salida:

% deColor(desvaste,blanco) deColor(ultramar,blanco)

De este modo, hemos conseguido los siguientes datos acerca de Ultramar y Desvaste:

- Desvaste es de color blanco.
- Ultramar es de color blanco.

Este conocimiento es consiste pues Desvaste era una caravana, luego se ha aplicado el defecto 1, y Ultramar era un vehículo marítimo, lo que quiere decir que se ha aplicado el defecto 5.

### Adición de un nuevo elemento a la información conocida.

Veamos qué información obtenemos acerca del color de un nuevo elemento, Levi, que es un vehículo no marítimo, según la información que se disponga sobre este.

**Ejemplo 1.**

Sólo conocemos que Levi es un vehículo no marítimo. Representamos esta información con las reglas:

```
elemento(levi).
es_un(levi,no_mar).
```

En la salida, no obtenemos información sobre el color de Levi, pues puede ser una excepción del defecto 2 (no hemos especificado que no es una caravana, autobús o autobús turístico).

**Ejemplo 2.**

Si especificamos ahora que Levi es un autobús:

```
es_un(levi,autobus).
```

Seguimos sin obtener información de su color, ya que no hemos especificado que no sea un autobús turístico.

**Ejemplo 3.**

Si ahora especificamos que es un autobús turístico:

```
es_un(levi,turistico).
```

Obtenemos:

% deColor(levi,azul)

Luego se obtiene que Levi es azul. Esto es gracias al defecto 4, que tiene prioridad respecto a los defectos 2 y 3.

Es decir, aunque anteriormente vimos que la prioridad entre defectos se podía representar mediante excepciones fuertes, en este caso, se ha representado a través de excepciones débiles (a través de estas, hemos representado que el defecto 4 tiene preferencia sobre el defecto 3 y que el defecto 3 tiene preferencia sobre el defecto 2).

De esta manera, se observa que la información más específica tiene preferencia sobre la información menos específica. En general, si tenemos C1 subclase de C2, y los defectos: "normalmente, los elementos de C2 tienen la propiedad P" y "normalmente, los elementos de C1 no tienen la propiedad P", entonces el segundo defecto predomina sobre el primero. Esto es lo que

se conoce como *principio de especificidad*.

# Capítulo 5

# El paradigma de programación ASP

En las secciones anteriores, nos hemos centrado en la modelización de bases de conocimiento en ASP con la finalidad de obtener elementos que verificasen ciertos predicados o para ver la veracidad o falsedad de ciertas afirmaciones. En esta sección, se pretende mostrar cómo se pueden usar las bases de conocimiento en ASP para encontrar soluciones a distintos problemas reduciéndolos a encontrar conjuntos de respuestas de programas de ASP. El procedimiento de resolución de problemas a través de reducirlos a encontrar conjuntos de respuesta de programas en ASP es lo que se denomina *el paradigma de programación ASP*.

En nuestro caso, vamos a construir bases de conocimiento cuyos conjuntos de respuesta den solución a los dos problemas siguientes: hallar los ciclos hamiltonianos de un grafo y encontrar la solución de sudokus.

## 5.1. Ciclos hamiltonianos de un grafo

Nuestro objetivo es construir un programa de ASP cuyos conjuntos de respuestas sean los ciclos hamiltonianos de un grafo dirigido G.

Un ciclo hamiltoniano de un grafo G es un camino que pasa por todos los vértices o nodos del grafo una sola vez y empieza y termina en el mismo vértice o nodo.

Para representar el grafo, se usan los predicados:

- **inicial/1**, donde *inicial(v0)* nos indica que el nodo $v_0$ es el nodo del que se parte (y por tanto, debe ser también el último nodo en visitarse).
- **nodo/1**, para representar los nodos del grafo.
- **arco/2**, donde *arco(v0,v1)* representa el arco del grafo G que parte del nodo $v_0$ y accede al nodo $v_1$.

Los ciclos hamiltonianos (que serán los conjuntos de respuesta del programa) se representan por conjuntos *en(v0,v1), en(v1,v2), ..., en(vk,v0)*, donde el predicado *en(vi,vj)* representa que el arco que parte de $v_i$ y accede a $v_j$ pertenece al ciclo hamiltoniano $< v0, v1, v2, ...vk, v0 >$ .

Para que un conjunto *en(v0,v1), en(v1,v2), ..., en(vk,v0)* forme un ciclo hamiltoniano de G, se debe cumplir:

1. Se accede y se sale de un vértice como máximo una vez.
2. Debe contener a todos los nodos del grafo.

Se representa que se accede como máximo una vez a cada vértice V del grafo de la siguiente manera: si el arco que parte desde uno de los nodos V1 del grafo y accede a V ya está en el ciclo hamiltoniano, es decir, se verifica *en(V1,V)*, entonces no puede existir otro vértice V2 distinto de V1 tal que el arco que parte de V2 y accede a V esté en el ciclo hamiltoniano. La regla que modeliza esta información es:

```
-en(V2,V) :- en(V1,V), V1 != V2, nodo(V1), nodo(V2), nodo(V).
```

De manera análoga, representamos que sólo se sale de un vértice V una única vez: si el arco que parte de V y llega a V1 está en el ciclo hamiltoniano, entonces cualquier otro arco que parta de V y llegue a un vértice V2 distinto a V1 no puede estar en el ciclo hamiltoniano. Esta información queda recogida por la regla:

```
-en(V,V2) :- en(V,V1), V1 != V2, nodo(V1), nodo(V2), nodo(V).
```

Representemos la otra condición que debe verificar un ciclo hamiltoniano: debe contener a todos los nodos del grafo.

Para representar este conocimiento, primero debemos definir la relación *alcanzable*: el nodo V es alcanzable en el ciclo que se esté considerando si existe un arco que parte del vértice inicial y accede a V o si en el ciclo existe un arco que parte de un vértice V1 alcanzable y accede a V. Este conocimiento se representa a través del predicado **alcanzable/1**, que es recursivo:

```
alcanzable(V) :- en(V1,V), inicial(V1), nodo(V).
alcanzable(V) :- en(V1,V), alcanzable(V1), nodo(V).
```

Para completar la información sobre los vértices que son y no son alcanzables, usamos la hipótesis del mundo cerrado: si no tenemos constancia de que un vértice sea alcanzable, entonces supondremos que no es alcanzable.

```
-alcanzable(V) :- not alcanzable(V), nodo(V).
```

De esta manera, se representa que el ciclo debe pasar por todos los nodos con la restricción siguiente, que impide la presencia de nodos no alcanzables en el conjunto de respuesta:

```
:- -alcanzable(V), nodo(V).
```

Con estas condiciones, CLINGO es capaz de determinar cuándo un ciclo es hamiltoniano y cuándo no.

Ahora, necesitamos representar los posibles candidatos a ciclos hamiltonianos, es decir, los posibles caminos que podemos formar en G. Para ello, siempre que exista un arco en G que parta de V1 y acceda a V2, se considerará la opción de incluirlo o no en el ciclo. Modelizamos esta información usando la disyunción:

```
en(V1,V2) | -en(V1,V2) :- arco(V1,V2).
```

Veamos algunos ejemplos de grafos y sus ciclos hamiltonianos.

**Ejemplo 1.**

Consideremos el grafo dirigido G de la figura 5.1.

Queremos encontrar los ciclos hamiltonianos que parten del nodo a.

Para describir a este grafo, usamos los predicados **nodo/1** y **arco/2**:

```
nodo(a;b;c;d;e).
arco(a,b;b,a;b,c;c,d;d,e;e,b;e,a;a,c).
```

y con el predicado **inicial/1**, indicamos el nodo del que queremos que parta:

```
inicial(a).
```

CLINGO nos devuelve para este grafo dos conjuntos de respuesta, que son los únicos ciclos hamiltonianos que posee el grafo:

%Answer: 1
% en(a,b) en(b,c) en(c,d) en(d,e) en(e,a)
%Answer: 2
% en(b,a) en(c,d) en(d,e) en(e,b) en(a,c)

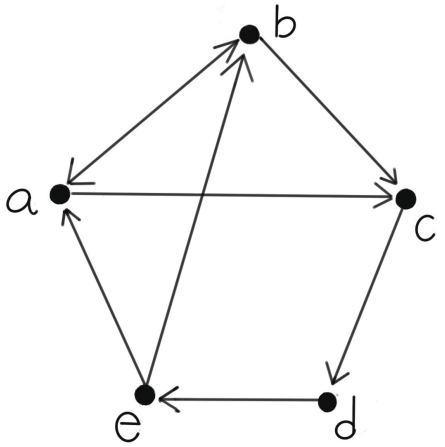



Figura 5.1: grafo G

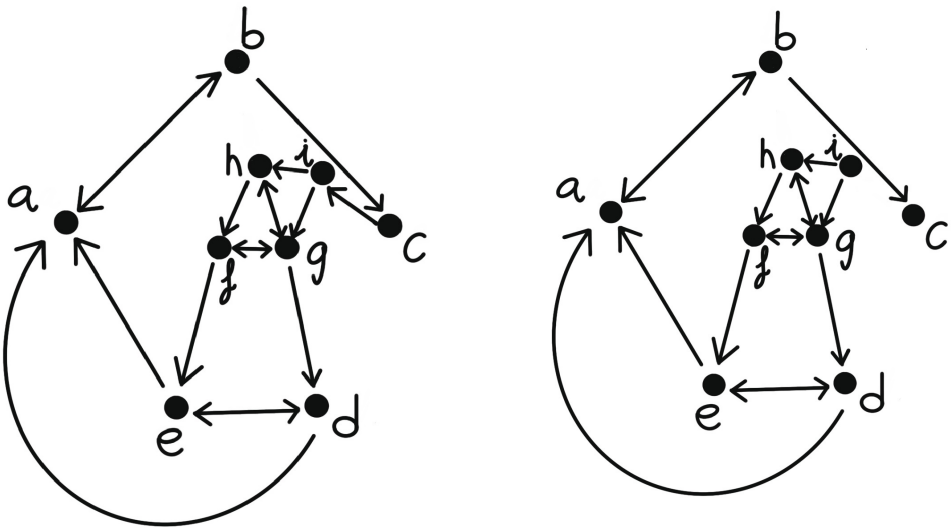
Vemos que ambos conjuntos satisfacen los requisitos para ser ciclos hamiltonianos: acceden y parten de todos los nodos una única vez y empiezan y terminan en el vértice a.

**Ejemplo 2.**

Consideremos los dos grafos de la figura 5.2.

De nuevo, queremos hallar los ciclos hamiltonianos de ambos grafos que parten del nodo a.

Estos dos grafos son el mismo salvo por un arco: el grafo $J_1$ posee un arco que parte del nodo c y accede al nodo i y el grafo $J_2$ no posee este arco. Esto hace que en $J_2$ no se pueda acceder a los nodos h, i, f, g, d y e, luego en este caso no debemos obtener ningún ciclo hamiltoniano.




(a) Grafo $J_1$ (b) Grafo $J_2$

Figura 5.2: grafos $J_1$ y $J_2$

Modelizamos el grafo $J_1$ a través de las reglas:

```
nodo(a;b;c;d;e;f;g;h;i).
arco(a,b;b,a;b,c;c,i;d,a;d,e;e,d;e,a;f,e;f,g;g,h;

     g,f;g,d;h,f;h,g;i,g;i,h).

inicial(a).
```

Para este grafo, obtenemos como solución tres ciclos hamiltonianos:

%Answer: 1
% en(a,b) en(b,c) en(c,i) en(d,a) en(e,d) en(f,e) en(g,f) en(h,g) en(i,h)
%Answer: 2
% en(a,b) en(b,c) en(c,i) en(d,a) en(e,d) en(f,e) en(g,h) en(h,f) en(i,g)
%Answer: 3
% en(a,b) en(b,c) en(c,i) en(d,e) en(e,a) en(f,g) en(g,d) en(h,f) en(i,h)

Modelizamos ahora el grafo $J_2$:

```
nodo(a;b;c;d;e;f;g;h;i).
arco(a,b;b,a;b,c;d,a;d,e;e,d;e,a;f,e;f,g;g,h;g,f;g,d;h,f;

     h,g;i,g;i,h).
```

```
inicial(a).
```

CLINGO nos devuelve:

%UNSATISFIABLE

En efecto, como dijimos, este grafo no posee ciclos hamiltonianos pues estos recorren todos los nodos del grafo y $J_2$ contiene algunos nodos a los que no se puede acceder.

Para generar los candidatos a ciclos hamiltonianos, también podemos modificar el programa anterior como sigue: se sustituye la última regla que acabamos de exponer, que se utiliza como hemos visto para generar los posibles caminos candidatos a ser ciclos hamiltonianos, por una regla que se denomina *regla de elección.*

Las *reglas de elección* tienen dos formas:

- $s_1 <_1 \{p(X) : q(X)\} <_2 s_2$ :- *cuerpo.*
- $s_1 <_1 \{p(c1); \ldots ; p(cr)\} <_2 s_2$ :- *cuerpo.*

donde $s_1$ y $s_2$ son enteros no negativos los cuales pueden omitirse y $<_1$ y $<_2$ son predicados de comparación.

La primera regla añade a los conjuntos de respuesta del programa ciertos conjuntos S formados por átomos $p(t)$ de manera que $s_1 \leq |S| \leq s_2$ y tales que si un átomo $p(t) \in S$, entonces el correspondiente $q(t)$ debe pertenecer al conjunto de respuesta. Esto es, la primera regla está generando un predicado $p$ en función de un predicado $q$ previamente definido. La segunda regla permite la adición a los conjuntos de repuesta de conjuntos S formados por átomos $p(c_i)$ entre los que se encuentran en la cabeza de la regla de manera que $s_1 \leq |S| \leq s_2$.

En nuestro caso, reemplazamos la última regla del programa para hallar los ciclos hamiltonianos de un grafo por la regla:

```
{en(V1,V2) : arco(V1,V2)}.
```

con la que se está definiendo el predicado **en/2** en función del predicado **arco/2** de forma que para cada arco de G, se plantea incluirlo o no en el ciclo que se esté considerando.

## 5.2. Sudokus

Un sudoku consiste en una cuadrícula con 9x9 celdas que se encuentra a su vez dividida en 3x3 regiones. El objetivo es rellenar todas las celdas con números del 1 al 9 partiendo de algunos números previamente colocados en algunas celdas de la cuadrícula. Las reglas para completar la cuadrícula son:

1. Cada celda debe contener un único número.
2. Ninguna fila ni columna puede contener un número repetido.
3. Ninguna de las regiones en las que se encuentra dividida la cuadrícula puede contener números repetidos.

Nuestro objetivo es construir un programa de ASP cuyos conjuntos de respuesta nos den las soluciones de cada sudoku que le sea proporcionado.

Para representar los números con los que se pueden rellenar las celdas del sudoku, usamos el predicado **num/1**:

```
num(1..9).
```

Las celdas se identifican con las coordenadas que ocupan en la cuadrícula. Para representarlas, usamos el predicado **crd/2**:

```
crd(X,Y) :- num(X), num(Y).
```

Como hemos dicho, la cuadrícula se encuentra dividida en 9 regiones. Cada coordenada (X,Y) pertenece a la región R, donde R es:

$$R = ((X-1)/3) * 3 + (Y+2)/3$$

y donde "$a/b$" representa la división entera de $a$ entre $b$.

De esta manera, representamos que la coordenada (X,Y) está en la región $R = ((X-1)/3) * 3 + (Y+2)/3$ de la cuadrícula mediante el predicado **estaEnRegion/3**:

```
estaEnRegion(X,Y,((X-1)/3)*3 + ((Y+2)/3)) :- crd(X,Y).
```

Se usa el predicado **pos/3** para indicar el número que se coloca en una determinada celda. De esta forma, el literal *pos(A,X,Y)* representa que en la celda que se identifica con la coordenada (X,Y) se encuentra colocado el número A.

En cada celda se puede poner un único número del 1 al 9. Para representar esta información, se usa la siguiente regla de elección:

```
1 {pos(A,X,Y) : num(A)} 1 :- crd(X,Y).
```

que genera todas las maneras posibles que hay de colocar un único número del 1 al 9 en la celda (X,Y). Es decir, hemos obtenido todas las posibles formas de rellenar la cuadrícula con números del 1 al 9 usando un único número en cada celda.

Ahora, debemos implementar las reglas del juego para que CLINGO pueda determinar qué cuadrículas son solución del sudoku y cuáles no lo son.

Primero, implementamos que ninguna fila puede contener un número repetido. Es decir, si existe una celda (X,Y1) en la que se ha colocado un número A, no puede haber otra celda distinta a la anterior en la misma fila, es decir, no puede haber otra celda de la forma (X,Y2) distinta a la anterior, en la que se coloque también el número A. Representamos esta información mediante la regla:

```
-pos(A,X,Y2) :- pos(A,X,Y1), Y1 != Y2, crd(X,Y1), crd(X,Y2),
                num(A).
```

Ahora debemos modelizar que ninguna columna puede contener un número repetido. Vamos a representar este conocimiento de la misma manera que se hizo antes: dada una coordenada (X1,Y) en la que se encuentre un número A, no puede existir otra coordenada (X2,Y) distinta a la anterior en la que se encuentre el mismo número A.

```
-pos(A,X2,Y) :- pos(A,X1,Y), X1 != X2, crd(X1,Y), crd(X2,Y),
                num(A).
```

Por último, debemos representar que ninguna región puede contener números repetidos. Lo representamos de manera similar a las otras restricciones: dada una región R, si tenemos una coordenada (X1,Y1) en la región en la que se haya colocado un determinado número A, no puede existir en la región otra coordenada distinta (X2,Y2) en la que se haya colocado el mismo número A.

```
-pos(A,X2,Y2) :- pos(A,X1,Y1),crd(X1,Y1) != crd(X2,Y2),
                 estaEnRegion(X1,Y1,R), estaEnRegion(X2,Y2,R),
                 num(A),crd(X1,Y1), crd(X2,Y2).
```

Para representar el sudoku que queramos resolver, se deben representar las celdas que vengan rellenas con números con el predicado **pos/3**.

Lo que realmente nos interesa de los conjuntos de respuesta que se obtienen son las posiciones, por tanto, usaremos la directriz #show:

```
#show pos/3.
```

Veamos ahora algunos ejemplos de sudokus.

**Ejemplo 1.**

Consideramos el sudoku de la figura 5.3.

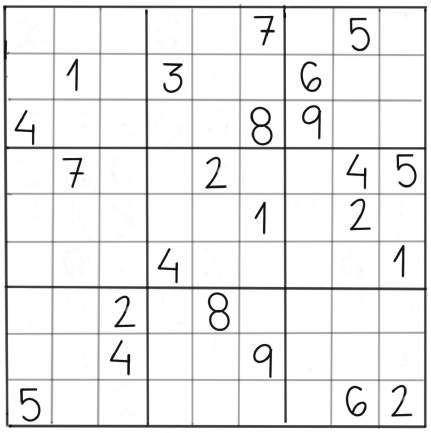


Figura 5.3: sudoku experto

Para describir el sudoku, como dijimos antes, usamos el predicado **pos/3**:

```
pos(7,1,6;5,1,8;1,2,2;3,2,4;6,2,7;4,3,1;8,3,6;9,3,7;7,4,2;
    2,4,5;4,4,8;5,4,9;1,5,6;2,5,8;4,6,4;1,6,9;2,7,3;8,7,5;
    4,8,3;9,8,6;5,9,1;6,9,8;2,9,9).
```

Obtenemos el conjunto de respuesta siguiente que da solución al problema:

%Answer: 1
%pos(7,1,6) pos(5,1,8) pos(1,2,2) pos(3,2,4) pos(6,2,7) pos(4,3,1)
%pos(8,3,6) pos(9,3,7) pos(7,4,2) pos(2,4,5) pos(4,4,8) pos(5,4,9)
%pos(1,5,6) pos(2,5,8) pos(4,6,4) pos(1,6,9) pos(2,7,3) pos(8,7,5)
%pos(4,8,3) pos(9,8,6) pos(5,9,1) pos(6,9,8) pos(2,9,9) pos(1,7,1)
%pos(1,4,3) pos(1,1,4) pos(1,8,5) pos(1,9,7) pos(1,3,8) pos(2,6,1)
%pos(2,3,2) pos(2,8,4) pos(2,2,6) pos(2,1,7) pos(3,5,1) pos(3,7,2)
%pos(3,1,3) pos(3,9,5) pos(3,6,6) pos(3,4,7) pos(3,8,8) pos(3,3,9)
%pos(4,5,2) pos(4,2,5) pos(4,9,6) pos(4,7,7) pos(4,1,9) pos(5,6,2)
%pos(5,2,3) pos(5,3,4) pos(5,5,5) pos(5,7,6) pos(5,8,7) pos(6,1,1)
%pos(6,8,2) pos(6,6,3) pos(6,7,4) pos(6,3,5) pos(6,4,6) pos(6,5,9)
%pos(7,8,1) pos(7,3,3) pos(7,9,4) pos(7,6,5) pos(7,5,7) pos(7,7,8)
%pos(7,2,9) pos(8,4,1) pos(8,1,2) pos(8,9,3) pos(8,5,4) pos(8,6,7)
%pos(8,2,8) pos(8,8,9) pos(9,2,1) pos(9,9,2) pos(9,5,3) pos(9,4,4)
%pos(9,1,5) pos(9,6,8) pos(9,7,9)

**Ejemplo 2.**

Consideramos ahora el sudoku de la figura 5.4.

Al igual que en el ejemplo anterior, usamos el predicado **pos/3** para describir el sudoku:

```
pos(3,2,1). pos(4,3,1). pos(8,3,2). pos(9,1,6). pos(8,2,4).
pos(2,2,5). pos(9,2,7). pos(7,3,9). pos(8,4,3). pos(7,5,3).
pos(9,6,2). pos(1,6,3). pos(9,4,5). pos(6,4,6). pos(1,5,5).
pos(5,6,5). pos(3,6,6). pos(7,4,7). pos(2,6,8). pos(6,9,2).
pos(1,8,4). pos(4,9,6). pos(4,7,8). pos(1,7,9). pos(9,9,8).
```

Obtenemos el siguiente conjunto de respuesta, que nos da la solución al sudoku:

%Answer: 1
%pos(3,2,1) pos(4,3,1) pos(8,3,2) pos(9,1,6) pos(8,2,4)
%pos(2,2,5) pos(9,2,7) pos(7,3,9) pos(8,4,3) pos(7,5,3)
%pos(9,6,2) pos(1,6,3) pos(9,4,5) pos(6,4,6) pos(1,5,5)
%pos(5,6,5) pos(3,6,6) pos(7,4,7) pos(2,6,8) pos(6,9,2)
%pos(1,8,4) pos(4,9,6) pos(4,7,8) pos(1,7,9) pos(9,9,8)
%pos(1,9,1) pos(1,2,2) pos(1,3,6) pos(1,1,7) pos(1,4,8)

%pos(2,4,1) pos(2,8,2) pos(2,1,3) pos(2,5,4) pos(2,7,6)
%pos(2,3,7) pos(2,9,9) pos(3,4,2) pos(3,7,3) pos(3,9,4)
%pos(3,3,5) pos(3,8,7) pos(3,5,8) pos(3,1,9) pos(4,5,2)
%pos(4,8,3) pos(4,4,4) pos(4,1,5) pos(4,6,7) pos(4,2,9)
%pos(5,5,1) pos(5,1,2) pos(5,9,3) pos(5,3,4) pos(5,8,6)
%pos(5,7,7) pos(5,2,8) pos(5,4,9) pos(6,6,1) pos(6,2,3)
%pos(6,1,4) pos(6,7,5) pos(6,5,7) pos(6,3,8) pos(6,8,9)
%pos(7,1,1) pos(7,7,2) pos(7,6,4) pos(7,9,5) pos(7,2,6)
%pos(7,8,8) pos(8,7,1) pos(8,8,5) pos(8,5,6) pos(8,9,7)
%pos(8,1,8) pos(8,6,9) pos(9,8,1) pos(9,3,3) pos(9,7,4)
%pos(9,5,9)

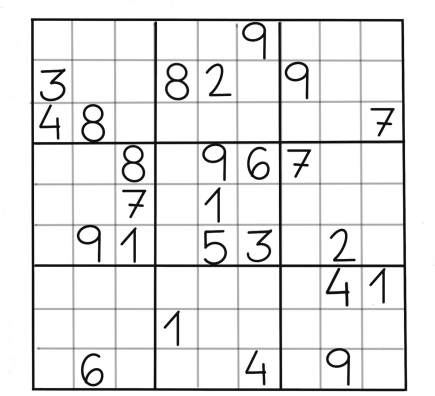


Figura 5.4: sudoku difícil

# Capítulo 6

# Aplicaciones

En este capítulo, vamos a estudiar tres puzzles que resultan de interés en la programación de conjuntos de respuesta, tanto por sus representaciones (en las que se utilizarán reglas de elección, restricciones, etc) como desde el punto de vista computacional. Estos tres puzzles son: el puzzle Nurikabe, por el que comenzaremos, el puzzle Heyawake y por último, el puzzle Masyu.

## 6.1. Puzzle Nurikabe

El Nurikabe consiste en una cuadrícula rectangular dividida en celdas, las cuales inicialmente son blancas y algunas de ellas están numeradas. El objetivo del juego es decidir qué casillas formarán el "nurikabe", es decir, qué casillas se pintan de negro, y cuáles formarán las "islas", esto es, cuáles permanecen blancas, de acuerdo a unas reglas que expondremos a continuación. Las casillas negras también se conocen como "agua" y las blancas, como "tierra".

Este puzzle es interesante tanto por su representación, que veremos a continuación, como por ser un problema NP-hard. Como se ha mencionado con anterioridad, los problemas de decisión P son los problemas que se pueden resolver en un tiempo polinomial, mientras que los NP son los problemas cuya solución puede ser verificada en un tiempo polinomial. Un problema se clasifica como NP-hard si cualquier problema de la clase NP se puede reducir a él en tiempo polinomial (aunque el problema es sí no tiene por qué pertenecer a la clase NP) y un problema se clasifica como NP-completo

si es NP y NP-hard al mismo tiempo. Se puede probar que el Nurikabe es un problema NP-hard, y decidir si existe solución del Nurikabe es un problema NP-completo. Para más información, puede visitarse el siguiente enlace: `enlace`.

Las reglas del Nurikabe son:

1. Todas las celdas numeradas deben permanecer blancas.
2. Toda celda blanca debe pertenecer a una isla.
3. Cada isla debe contener una única celda numerada.
4. Cada isla debe estar formada por el número de celdas blancas que indique la casilla numerada que contiene.
5. Las celdas blancas de cada isla deben estar ortogonalmente conectadas.
6. Dos islas no pueden estar conectadas.
7. Todas las celdas negras deben estar conectadas ortogonalmente.
8. Ningún subconjunto de celdas negras puede formar un cuadrado 2x2.

Se puede visitar también el siguiente enlace para encontrar más información sobre las reglas del Nurikabe: `enlace`. Además, este nos permite jugar en línea, lo que puede resultar de utilidad para generar más ejemplos de puzzles Nurikabe aparte de los que vamos a exponer.

Comencemos pues con la modelización de los puzzles Nurikabe.

Supongamos que nos dan una cuadrícula rectangular de dimensión nxm (n filas y m columnas). Se representa la cuadrícula describiendo las celdas que están numeradas con el predicado **numerada/3**, donde *numerada(X,Y,A)* se verifica si la celda que ocupa la posición (X,Y) en la cuadrícula, con X la fila e Y la columna, contiene el número A (por tanto, esta celda va a pertener a una isla de tamaño A).

Para representar las filas y las columnas de la cuadrícula, se usa el predicado **fila/1** y **col/1**:

```
fila(1..n).
col(1..m).
```

Se representan los dos colores de celdas que podemos tener con el predicado **color/1**, donde $b$ representa el color blanco y $n$ representa el color negro:

```
color(n).
color(b).
```

Vamos a generar todas las posibles cuadrículas que se pueden considerar seleccionando subconjuntos de celdas que permanecerán blancas y pintando de negro las celdas restantes.

Para generar las cuadrículas, se usa el predicado **celda/3**, de forma que *celda(C,X,Y)* representa que la celda que está en la posición (X,Y) es de color C. A través de una regla de elección, se generan los conjuntos de celdas que permanecerán blancas:

```
{celda(b,X,Y)} 1 :- fila(X), col(Y).
```

Para pintar las restantes celdas de negro, usamos la siguiente regla:

```
celda(n,X,Y) :- not celda(b,X,Y), fila(X), col(Y).
```

Ahora, vamos a implementar las reglas para eliminar las cuadrículas que no las verifiquen.

**REGLA 1.**

Debemos modelizar la regla 1: "todas las celdas numeradas deben permanecer blancas". Representamos la regla con una restricción, de manera que para verificarse no puede ocurrir que una celda (X,Y) sea negra y a su vez esté numerada:

```
:- numerada(X,Y,A), celda(n,X,Y).
```

**REGLAS 2 Y 5.**

Para la implementación de 2 y 5, usamos la regla 3: por las reglas 2, 3 y 5, sabemos que cada celda blanca debe pertenecer a una isla, que cada isla debe contener una única celda numerada y que las celdas blancas de cada isla se conectan ortogonalmente. Por tanto, toda celda blanca debe estar conectada ortogonalmente a una celda numerada.

Para representar este conocimiento, vamos a definir antes algunos conceptos.

Primero, vamos a definir la relación "ser adyacentes": las celdas adyacentes a una celda (X,Y) son las celdas (X+1,Y), (X-1,Y), (X,Y+1), (X,Y-1). Representamos que la celda (X,Y) y la celda (U,V) son adyacentes usando el predicado **ady/4:**

```
ady(X,Y,U,V) :- fila(X), fila(U), col(Y), col(V),
                |X-U|+|Y-V| == 1.
```

*Nota: también se puede denominar ortogonal, pues representa que dos celdas están conectadas ortogonalmente.*

Proseguimos modelizando cuándo dos celdas se van a conectar ortogonalmente a través de un color.

Para ello, introducimos el predicado **conectadas/4**, que se verifica si:

1. *conectadas(C,X,Y,X,Y)* se verifica si (X,Y) es una celda de color C. Es decir, vamos a considerar que una celda (X,Y) de color C está conectada consigo misma a través del color C.
2. *conectadas(C,X,Y,U,V)* se verifica si la celda (U,V) es de color C y si existe una celda intermedia que es adyacente a (U,V) y está conectada ortogonalmente a través del color C a (X,Y).

El predicado está definido por recursión, así que lo representamos con las reglas:

```
conectadas(C,X,Y,X,Y) :- celda(C,X,Y).
conectadas(C,X,Y,U,V) :- celda(C,U,V), conectadas(C,X,Y,X1,Y1),
                         ady(X1,Y1,U,V).
```

Finalmente, para representar que toda celda blanca debe estar conectada ortogonalmente a una celda numerada, se define el predicado **conectadaB/2**, donde *conectadaB(X,Y)* se verifica si (X,Y) está conectada ortogonalmente a través del color blanco a una celda numerada, e imponemos que toda celda blanca verifique el predicado **conectadaB/2** a través de una restricción:

```
conectadaB(X,Y) :- conectadas(b,X,Y,U,V), celda(b,X,Y),
                   numerada(U,V,A).
:- celda(b,X,Y), not conectadaB(X,Y).
```

**REGLAS 3 Y 6.**

Para representar las reglas 6 y 3, usamos la regla 2: se tiene que cada isla contiene una única celda numerada, que toda celda blanca pertenece a una isla y que dos islas no pueden estar conectadas. Esto implica que dos celdas numeradas no van poder estar conectadas ortogonalmente a través del color blanco porque sino, dos islas distintas estarían conectadas.

```
:- conectadas(b,X,Y,U,V), numerada(X,Y,A), numerada(U,V,M),
    (X,Y) != (U,V).
```

**REGLA 4.**

Para la regla 4: "cada isla debe estar formada por el número de celdas blancas que indique la casilla numerada que contiene", vamos a definir qué se considera por isla.

La isla estará formada por una casilla numerada (X,Y), supongamos que es tal que se verifica *numerada(X,Y,A)*, y por exactamente A celdas blancas que se conectan ortogonalmente a través del color blanco a la celda numerada. Es decir, el cardinal del conjunto de celdas blancas que se conectan ortogonalmente a (X,Y) debe ser exactamente A. Como por la regla 3, cada isla contiene una única celda numerada, identificamos la isla con la celda numerada:

```
isla(X,Y) :- numerada(X,Y,A), A {conectadas(b,X,Y,U,V)} A,
             fila(X), col(Y).
```

Todas las celdas numeradas deben definir una isla:

```
:- numerada(X,Y,A), not isla(X,Y).
```

**REGLA 7.**

Veamos cómo representar la regla 7: "todas las celdas negras deben estar conectadas ortogonalmente". Para ello, usamos de nuevo una restricción: no pueden existir dos celdas (X,Y) y (U,V) de color negro que no estén conec-

tadas ortogonalmente a través del color negro.

```
:- celda(n,X,Y), celda(n,U,V), not conectadas(n,X,Y,U,V).
```

**REGLA 8.**

Por último, representemos la regla 8: "ningún subconjunto de celdas negras puede formar un cuadrado 2x2". Para ello, definimos primero cuándo un subconjunto de celdas negras define un cuadrado: si tenemos una celda (X,Y) negra, está define un cuadrado cuando las celdas (X,Y+1), (X+1,Y) y (X+1,Y+1) también son negras.

```
cuadradoN(X,Y) :- celda(n,X,Y), celda(n,X,Y+1), celda(n,X+1,Y),
                  celda(n,X+1,Y+1).
```

Finalmente, se impide que cualquier celda forme un cuadrado de celdas negras usando una restricción:

```
:- cuadradoN(X,Y).
```

Como al principio todas las celdas de la cuadrícula son blancas, la solución del Nurikabe será el conjunto de celdas que finalmente se han pintado de negro. Así, definimos el predicado **celdaNegra/2**, que se verifica si la celda está pintada de negro:

```
celdaNegra(X,Y) :- celda(n,X,Y).
```

Por tanto, la solución a nuestro problema nos la dará el conjunto de literales de la forma *celdaNegra(X,Y)* que aparezca en el conjunto de respuesta:

```
#show celdaNegra/2.
```

Veamos ahora algunos ejemplos de puzzles Nurikabe.

**Ejemplo 1.**

En este ejemplo, vamos a considerar el Nurikabe de la figura 6.1.

Vamos a describir este puzzle con usando el predicado **numerada/3**:

```
numerada(2,1,2). numerada(3,3,2). numerada(3,5,4). numerada(5,2,2).
numerada(5,6,3). numerada(6,4,2). numerada(8,4,3). numerada(9,2,6).
numerada(10,7,4).
```

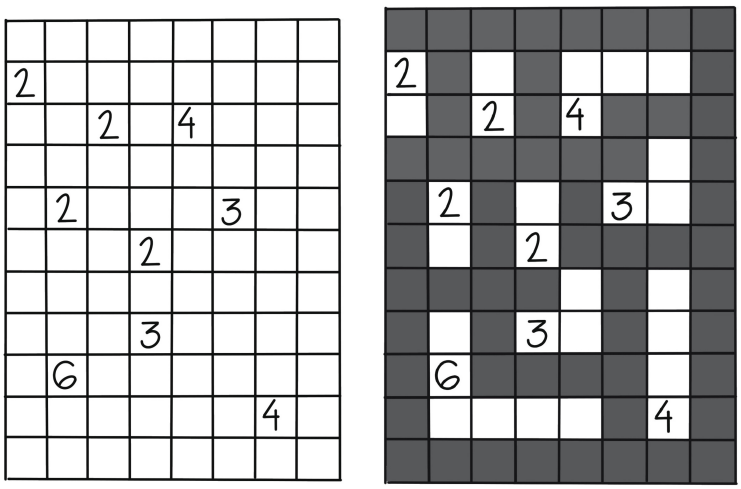
Para cargar el fichero, debemos especificarle a CLINGO el número de

filas y columnas de la cuadrícula (lo que denominamos como n y m, respectivamente). Guardando el código del Nurikabe en el archivo `nurikabe.lp` y el fichero con la descripción de la cuadrícula en el archivo con nombre `nurikabe1.lp`, cargamos los archivos como:

```
clingo nurikabe.lp nurikabe1.lp -c n=11 -c m=8
```

y nos devuelve la solución del puzzle:

%celdaNegra(1,1) celdaNegra(4,1) celdaNegra(5,1) celdaNegra(6,1)
%celdaNegra(7,1) celdaNegra(8,1) celdaNegra(9,1) celdaNegra(10,1)
%celdaNegra(11,1) celdaNegra(1,2) celdaNegra(2,2) celdaNegra(3,2)
%celdaNegra(4,2) celdaNegra(7,2) celdaNegra(11,2) celdaNegra(1,3)
%celdaNegra(4,3) celdaNegra(5,3) celdaNegra(6,3) celdaNegra(7,3)
%celdaNegra(8,3) celdaNegra(9,3) celdaNegra(11,3) celdaNegra(1,4)
%celdaNegra(2,4) celdaNegra(3,4) celdaNegra(4,4) celdaNegra(7,4)
%celdaNegra(9,4) celdaNegra(11,4) celdaNegra(1,5) celdaNegra(4,5)
%celdaNegra(5,5) celdaNegra(6,5) celdaNegra(9,5) celdaNegra(11,5)
%celdaNegra(1,6) celdaNegra(3,6) celdaNegra(4,6) celdaNegra(6,6)
%celdaNegra(7,6) celdaNegra(8,6) celdaNegra(9,6) celdaNegra(10,6)
%celdaNegra(11,6) celdaNegra(1,7) celdaNegra(3,7) celdaNegra(6,7)
%celdaNegra(11,7) celdaNegra(1,8) celdaNegra(2,8) celdaNegra(3,8)
%celdaNegra(4,8) celdaNegra(5,8) celdaNegra(6,8) celdaNegra(7,8)
%celdaNegra(8,8) celdaNegra(9,8) celdaNegra(10,8) celdaNegra(11,8)



(a) Nurikabe nivel medio (b) Solución

Figura 6.1: puzzle Nurikabe y su solución

**Ejemplo 2.**

Estudiemos ahora la solución del Nurikabe de la figura 6.2.

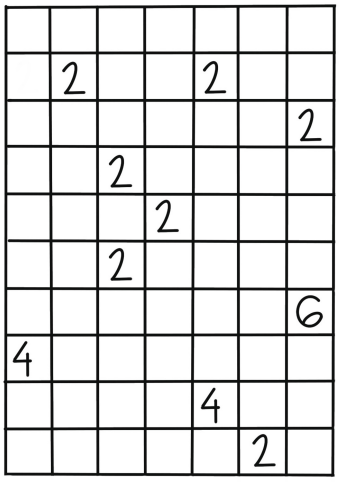



Figura 6.2: puzzle Nurikabe nivel medio

Describamos el puzzle de nuevo con el predicado **numerada/3:**

```
numerada(2,2,2). numerada(2,5,2). numerada(3,7,2). numerada(4,3,2).
numerada(5,4,2). numerada(6,3,2). numerada(7,7,6). numerada(8,1,4).
numerada(9,5,4). numerada(10,6,2).
```

Ahora, guardamos el código de la descripción en el fichero `nurikabe2.lp`. Cargando de nuevo los archivo como:

```
clingo nurikabe.lp nurikabe2.lp -c n=10 -c m=7
```

nos devuelve la solución del Nurikabe:

%celdaNegra(1,1) celdaNegra(2,1) celdaNegra(3,1) celdaNegra(4,1)
%celdaNegra(5,1) celdaNegra(6,1) celdaNegra(1,2) celdaNegra(3,2)
%celdaNegra(5,2) celdaNegra(7,2) celdaNegra(8,2) celdaNegra(9,2)
%celdaNegra(10,2) celdaNegra(1,3) celdaNegra(3,3) celdaNegra(5,3)
%celdaNegra(7,3) celdaNegra(10,3) celdaNegra(1,4) celdaNegra(2,4)
%celdaNegra(3,4) celdaNegra(4,4) celdaNegra(6,4) celdaNegra(7,4)
%celdaNegra(8,4) celdaNegra(10,4) celdaNegra(1,5) celdaNegra(4,5)
%celdaNegra(6,5) celdaNegra(8,5) celdaNegra(10,5) celdaNegra(1,6)
%celdaNegra(2,6) celdaNegra(3,6) celdaNegra(4,6) celdaNegra(5,6)

%celdaNegra(6,6) celdaNegra(8,6) celdaNegra(9,6) celdaNegra(1,7)
%celdaNegra(4,7) celdaNegra(9,7)

## 6.2. Puzzle Heyawake

El Heyawake es un puzzle que se juega en una cuadrícula rectangular dividida en celdas. Además, la cuadrícula se divide a su vez en habitaciones rectangulares de diversos tamaños. Al comienzo, las celdas de la cuadrícula son blancas y algunas habitaciones contienen una celda numerada. El objetivo es decidir qué celdas deben permanecer blancas y qué celdas se pintan de negro de acuerdo a las reglas que se exponen a continuación. Al igual que el Nurikabe, decidir si existe una solución del puzzle Heyawake es un problema NP-completo.

Las reglas del puzzle son las siguientes:

1. Dos celdas negras no pueden ser adyacentes ni vertical ni horizontalmente.
2. Todas las celdas blancas deben estar conectadas ortogonalmente.
3. En las habitaciones en las que se encuentre una celda numerada, el número de la celda indica cúantas celdas de la habitación deben pintarse de negro.
4. Una habitación sin celda numerada puede contener cualquier número de celdas negras.
5. Un camino recto de celdas blancas no puede atravesar más de dos habitaciones.

De nuevo, podemos visitar el siguiente enlace para obtener más información sobre las reglas del juego y poder jugar en línea: `enlace`.

Veamos entonces cómo se puede modelizar el puzzle. Primero, vamos a describir la cuadrícula dada.

Supongamos que la cuadrícula que consideramos tiene dimensiones nxm (n filas y m columnas) y está dividida en r habitaciones.

Para describir la cuadrícula, a cada habitación se le asigna una etiqueta y se representa a través del predicado **hab/4**: supongamos que tenemos una habitación delimitada por las celdas (X1,Y1), (X1,Y2), (X2,Y1) y (X2,Y2) a la que se le asigna una etiqueta A, entonces, esto se representa como *hab(A,X1,Y1,X2,Y2)*. La etiqueta A que podemos asignarle a cada habitación irá de 0 a r-1, y se representa por el predicado **etiqueta/1**.

```
fila(1..n).
col(1..m).
etiqueta(0..r-1).
```

Algunas de las habitaciones contienen una celda numerada y otras no. Para representarlo, usamos el predicado **contiene/2**, de manera que *contiene(A,N)* representa que la habitación cuya etiqueta es A contiene una celda numerada con el número N. Para representar que la habitación A no tiene celda numerada, se usa *contiene(A,-1)*.

Estudiemos entonces el programa que nos dará la solución del Heyawake.

Vamos a generar las distintas cuadrículas candidatas a ser solución atendiendo a las reglas 3 y 4, contemplando todas las maneras de pintar celdas de negro en cada habitación de la cuadrícula (la celda numerada también puede pintarse de negro esta vez).

Para ello, definimos primero el "tamaño de la habitación": una habitación con una celda numerada con el número N tendrá tamaño N, y una habitación sin celda numerada tendrá tamaño -1. El tamaño de la habitación nos indica cuántas celdas de la habitación pueden pintarse de negro. Usamos el predicado **dimHab/4** para representar el tamaño de la habitación:

```
dimHab(N,X1,Y1,X2,Y2) :- hab(A,X1,Y1,X2,Y2), contiene(A,N).
```

**REGLAS 3 Y 4.**

Veamos cómo representar las reglas 3 y 4. Para ello, usamos el predicado **celdaNegra/2** donde *celdaNegra(X,Y)* indica, como antes, que la celda (X,Y), donde X es la fila e Y la columna, se ha pintado de negro. Usamos también el predicado **dentrohab/3**, donde *dentrohab(X,Y,A)* representa que la celda (X,Y) está dentro de la habitación A:

```
dentrohab(X,Y,A) :- hab(A,X1,Y1,X2,Y2), X1<=X, X<=X2, Y1<=Y,
                    Y<=Y2, fila(X), col(Y).
```

Generemos todas las posibles maneras de pintar celdas de negro en una habitación que contiene una celda numerada atendiendo a la regla 3. Usamos una regla de elección:

```
N {celdaNegra(X,Y):dentrohab(X,Y,A)} N :- hab(A,X1,Y1,X2,Y2),
                                          dimHab(N,X1,Y1,X2,Y2),
                                          N>0.
```

Ahora, generamos todas las posibles formas de pintar celdas de negro en una habitación que no tiene ninguna celda numerada de acuerdo a la regla 4.

```
{celdaNegra(X,Y):dentrohab(X,Y,A)} :- hab(A,X1,Y1,X2,Y2),
                                      dimHab(-1,X1,Y1,X2,Y2).
```

Las celdas restantes que no hayamos pintado de negro permanecerán blancas:

```
celdaBlanca(X,Y) :- not celdaNegra(X,Y), fila(X), col(Y).
```

Implementemos las reglas que quedan para descartar las cuadrículas que no las verifican.

**REGLA 1.**

Tenemos que modelizar la regla 1: "dos celdas negras no pueden ser adyacentes ni vertical ni horizontalmente".

Como ya hicimos, usamos el predicado **ady/4** para representar que dos celdas son adyacentes. Dada una celda (X,Y), sus adyacentes son las celdas (X+1,Y), (X-1,Y),(X,Y+1), (X,Y-1). Por tanto, definimos el predicado con la regla:

```
ady(X1,Y1,X2,Y2) :- fila(X1), fila(X2), col(Y1), col(Y2),
                    |X1-X2| + |Y1-Y2| == 1.
```

Representamos que dos celdas negras no pueden ser adyacentes ni vertical ni horizontalmente a través de la restricción:

```
:- ady(X,Y,X1,Y1), celdaNegra(X,Y), celdaNegra(X1,Y1),
   (X,Y) != (X1,Y1).
```

**REGLA 2.**

La regla 2 nos indica que todas las celdas blancas deben estar conectadas ortogonalmente. Para representarlo, usamos como en el puzzle anterior el predicado **conectadas/4**, donde esta vez:

- *conectadas(X,Y,X1,Y1)* se verifica si las celdas son blancas y adyacentes.
- *conectadas(X,Y,X1,Y1)* se verifica si (X,Y) es adyacente a (W,Z), (W,Z) está conectada ortogonalmente a (X1,Y1), y tanto (X,Y) como (W,Z) y (X1,Y1) son blancas.

Así, modelizamos la regla 2 a través de las siguientes reglas:

```
conectadas(X,Y,X1,Y1) :- celdaBlanca(X,Y), celdaBlanca(X1,Y1),
                         ady(X,Y,X1,Y1).
conectadas(X,Y,X1,Y1) :- celdaBlanca(X,Y), ady(X,Y,W,Z),
                         conectadas(W,Z,X1,Y1).
```

Finalmente, se representa que todas las celdas blancas están conectadas ortogonalmente mediante una restricción:

```
:- not conectadas(X,Y,X1,Y1), celdaBlanca(X,Y), celdaBlanca(X1,Y1),
   (X,Y)!=(X1,Y1).
```

**REGLA 5.**

Por último, representemos la regla 5: "un camino recto de celdas blancas no puede atravesar más de dos habitaciones".

Con este fin, definimos primero algunos conceptos.

El camino recto de celdas blancas formará un segmento horizontal o vertical. Representamos que el segmento es horizontal o vertical con el predicado **seg/1**, donde *seg(h)* se verifica si el segmento es horizontal y *seg(v)* si el segmento es vertical.

```
seg(h).
seg(v).
```

Definamos cuándo una camino recto de celdas blancas atraviesa 1, 2 o 3 habitaciones. Lo hacemos con el predicado **camino/6** como sigue:

1. Siempre que tengamos una celda blanca, se va a considerar que forma un camino tanto horizontal como vertical que atraviesa una única habitación.

   ```
   camino(S,X,Y,X,Y,1) :- celdaBlanca(X,Y), seg(S).
   ```

2. Sea (X,Y) una celda. Entonces, existe un camino horizontal de celdas blancas entre (X,Y) y una celda de su misma fila, (X,Y1), que atraviesa N habitaciones con N $<$ 3 si (X,Y) es blanca y la celda (X,Y+1), que es adyacente a (X,Y), está en la misma habitación que (X,Y) y entre ella y la celda (X,Y1) existe un camino horizontal de celdas blancas que atraviesa N habitaciones.

   ```
   camino(h,X,Y,X,Y1,N) :- celdaBlanca(X,Y), dentrohab(X,Y,A),
                           dentrohab(X,Y+1,A), N<3,
                           camino(h,X,Y+1,X,Y1,N).
   ```

3. Si (X,Y) es una celda blanca, existirá un camino horizontal de celdas blancas entre (X,Y) y (X,Y1) que atraviesa N+1 habitaciones con N $<$ 3 si la celda (X,Y+1) no está en la misma habitacíon que (X,Y) y verifica que existe un camino horizontal de celdas blancas que atraviesa N habitaciones entre ella y (X,Y1).

   ```
   camino(h,X,Y,X,Y1,N+1) :- celdaBlanca(X,Y), dentrohab(X,Y,A),
                             dentrohab(X,Y+1,B), A!=B, N<3,
                             camino(h,X,Y+1,X,Y1,N).
   ```

4. De manera análoga, existe un camino vertical de celdas blancas entre (X,Y) y una celda de su misma columna, (X1,Y), que atraviesa N habitaciones con N $<$ 3 si (X,Y) es una celda blanca y la celda (X+1,Y), que es adyacente a (X,Y), está en la misma habitación que (X,Y) y verifica que existe un camino de celdas blancas vertical que atraviesa N habitaciones entre ella y (X1,Y).

   ```
   camino(v,X,Y,X1,Y,N) :- celdaBlanca(X,Y), dentrohab(X,Y,A),
                           dentrohab(X+1,Y,A), N<3,
                           camino(v,X+1,Y,X1,Y,N).
   ```

5. Existe un camino vertical de celdas blancas entre (X,Y) y (X1,Y) que atraviesa N+1 habitaciones con N $< 3$ si la celda (X,Y) es blanca, la celda (X+1,Y) no está en la misma habitación que (X,Y) y verifica que entre ella y (X1,Y) existe un camino vertical de celdas blancas que atraviesa N habitaciones.

```
camino(v,X,Y,X1,Y,N+1) :- celdaBlanca(X,Y), dentrohab(X,Y,A),
                          dentrohab(X+1,Y,B), A!=B, N<3,
                          camino(v,X+1,Y,X1,Y,N).
```

Finalmente, para modelizar la regla 5, impedimos que las celdas blancas formen caminos atravesando 3 habitaciones (ya que si atraviesan más de dos habitaciones, en particular atraviesan 3) mediante una restricción:

```
:- camino(S,X,Y,X1,Y1,3).
```

De igual forma que en el Nurikabe, como al principio las celdas son blancas, nos interesa saber qué celdas se han pintado de negro. Para eso, usamos la directriz #show:

```
#show celdaNegra/2.
```

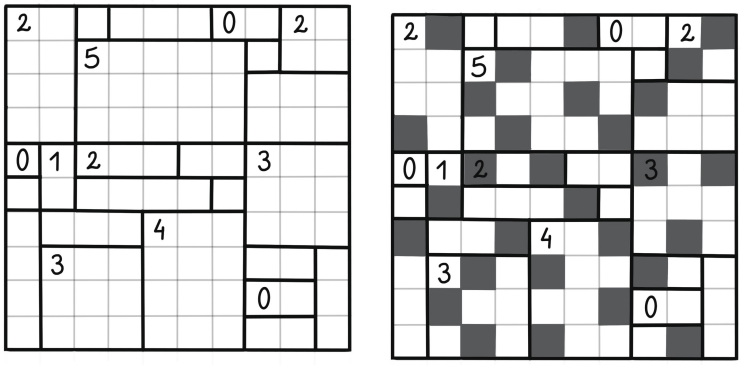
Procedamos a ver algunos ejemplos.

**Ejemplo 1.**

Sea el Heyawake de la figura 6.3.

Describimos la cuadrícula usando los predicados **hab/5** y **contiene/2** como mencionamos anteriormente:

```
hab(0,1,1,4,2). hab(1,1,3,1,3). hab(2,1,4,1,6). hab(3,1,7,1,8).
hab(4,1,9,2,10).hab(5,2,3,4,7). hab(6,2,8,2,8). hab(7,3,8,4,10).
hab(8,5,1,5,1). hab(9,5,2,6,2). hab(10,5,3,5,5). hab(11,5,6,5,7).
hab(12,5,8,7,10). hab(13,6,1,6,1). hab(14,6,3,6,6).
hab(15,6,7,6,7). hab(16,7,1,10,1). hab(17,8,2,10,4).
hab(18,7,5,10,7). hab(19,8,8,8,9). hab(20,9,8,9,9).
hab(21,10,8,10,9). hab(22,8,10,10,10). hab(23,7,2,7,4).
contiene(0,2). contiene(1,-1). contiene(2,-1). contiene(3,0).
contiene(4,2). contiene(5,5). contiene(6,-1). contiene(7,-1).
contiene(8,0). contiene(9,1). contiene(10,2). contiene(11,-1).
```



(a) Heyawake nivel normal (b) Solución

Figura 6.3: puzzle Heyawake y su solución

```
contiene(12,3). contiene(13,-1). contiene(14,-1). contiene(15,-1).
contiene(16,-1). contiene(17,3). contiene(18,4). contiene(19,-1).
contiene(20,0). contiene(21,-1). contiene(22,-1). contiene(23,-1).
```

Guardamos el código en el archivo `heyawake.lp` y la descripción de la cuadrícula en `heyawakeNormal.lp`. Ejecutando CLINGO:

```
clingo heyawake.lp heyawakeNormal.lp -c n=10 -c m=10 -c r=24
```

obtenemos la solución al puzzle:

%celdaNegra(4,1) celdaNegra(7,1) celdaNegra(1,2) celdaNegra(6,2)
%celdaNegra(9,2) celdaNegra(3,3) celdaNegra(5,3) celdaNegra(8,3)
%celdaNegra(10,3) celdaNegra(2,4) celdaNegra(4,4) celdaNegra(7,4)
%celdaNegra(5,5) celdaNegra(8,5) celdaNegra(10,5) celdaNegra(1,6)
%celdaNegra(3,6) celdaNegra(6,6) celdaNegra(4,7) celdaNegra(7,7)
%celdaNegra(9,7) celdaNegra(3,8) celdaNegra(5,8) celdaNegra(8,8)
%celdaNegra(2,9) celdaNegra(7,9) celdaNegra(10,9) celdaNegra(1,10)
%celdaNegra(5,10)

**Ejemplo 2.**

Buscamos la solución del Heyawake de la figura 6.4.

Este puzzle se representa por las siguientes reglas:

```
hab(0,1,1,1,2). hab(1,1,3,2,5). hab(2,1,6,7,7).
hab(3,1,8,2,8). hab(4,1,9,4,10). hab(5,2,1,4,2).
hab(6,3,3,5,3). hab(7,3,4,3,5). hab(8,4,4,5,4).
hab(9,4,5,5,5). hab(10,3,8,7,8). hab(11,5,9,5,10).
hab(12,6,9,7,10). hab(13,5,1,7,2). hab(14,6,3,6,5).
hab(15,7,3,7,5). hab(16, 8,1,8,3). hab(17,9,1,10,1).
hab(18,9,2,10,3). hab(19,8,4,10,6). hab(20,8,7,8,9).
hab(21,9,7,9,7). hab(22,9,8,9,8). hab(23,9,9,9,9).
hab(24,8,10,10,10). hab(25,10,7,10,9).
contiene(0,-1). contiene(1, 2). contiene(2,4). contiene(3,-1).
contiene(4,-1). contiene(5,-1). contiene(6,2). contiene(7,1).
contiene(8,-1). contiene(9,0). contiene(10,2). contiene(11,-1).
contiene(12,2). contiene(13,2). contiene(14,-1).
contiene(15,-1). contiene(16,-1). contiene(17,-1).
contiene(18,0). contiene(19,-1). contiene(20,-1).
contiene(21,0). contiene(22,-1). contiene(23,1).
contiene(24,0). contiene(25,-1).
```

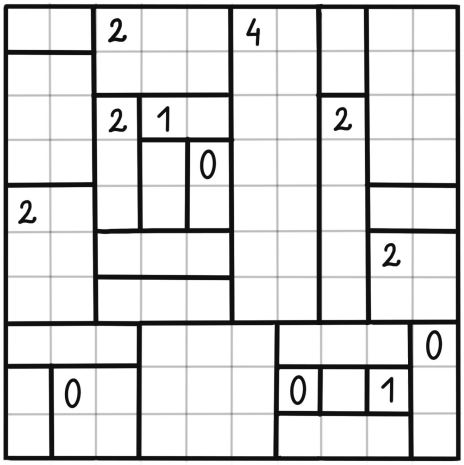



Figura 6.4: puzzle Heyawake nivel difícil

Guardando esta descripción en el archivo `heyawakeDificil.lp` y ejecutando CLINGO:

```
clingo heyawake.lp heyawakeDificil.lp -c n=10 -c m=10 -c r=26
```

obtenemos la solución al puzzle:

%celdaNegra(5,1) celdaNegra(9,1) celdaNegra(2,2) celdaNegra(7,2)
%celdaNegra(1,3) celdaNegra(3,3) celdaNegra(5,3) celdaNegra(8,3)
%celdaNegra(4,4) celdaNegra(7,4) celdaNegra(10,4) celdaNegra(1,5)
%celdaNegra(3,5) celdaNegra(6,5) celdaNegra(5,6) celdaNegra(9,6)
%celdaNegra(2,7) celdaNegra(4,7) celdaNegra(7,7) celdaNegra(10,7)
%celdaNegra(3,8) celdaNegra(5,8) celdaNegra(8,8) celdaNegra(1,9)
%celdaNegra(6,9) celdaNegra(9,9) celdaNegra(4,10) celdaNegra(7,10)

## 6.3. Puzzle Masyu

El puzzle Masyu (conocido también como Pearl Puzzle) se juega en una cuadrícula rectangular. Algunas de las celdas de la cuadrícula contienen un círculo blanco o negro. El objetivo es pintar una única línea continua y sin intersecciones que pase por todos los círculos de la cuadrícula una única vez, atendiendo a las siguientes reglas:

1. La línea debe formar un único camino cerrado, de modo que si atraviesa una celda, debe acceder y salir de la celda una única vez.
2. El camino que estamos construyendo debe atravesar las celdas con círculos blancos en línea recta pero debe girar 90 grados en la celda que se recorre justo antes a la celda que contiene el círculo blanco o en la que se recorre justo después.
3. El camino debe realizar giros de 90 grados en las celdas que contengan círculos negros pero debe atravesar tanto la celda posterior en el camino como la anterior en el camino a la celda que contiene el círculo negro en línea recta.

De nuevo, al igual que en los puzzles Nurikabe y Heyawake, decidir cuándo existe una solución del puzzle y cuándo no constituye un problema NP-completo. Para más información sobre las reglas del juego y para poder jugar en línea se puede visitar el enlace: `enlace`. Dicho esto, comencemos con la modelización.

Supongamos de nuevo que nos dan una cuadrícula nxm. Para representar la cuadrícula, debemos describir qué celdas contienen círculos blancos y qué celdas contienen círculos negros. Usamos para esto los predicados **negro/2**

y **blanco/2**, donde *negro(X,Y)* nos indica que la celda que se encuentra en la fila X y en la columna Y contiene un círculo negro y, análogamente, *blanco(X,Y)* nos indica que la celda (X,Y) contiene un círculo blanco.

Veamos ahora como generar la línea solución del puzzle. Para representar las filas y las columnas de la cuadrícula, se usa como anteriormente los predicados **col/1** y **fila/1**:

```
col(1..m).
fila(1..n).
```

Ahora, vamos a considerar todos los segmentos posibles que podemos tener atravesando las celdas del tablero. Las direcciones que pueden llevar estos segmentos son la horizontal y la vertical, y lo representamos con el predicado **direc/1**, donde *direc(h)* representa que el segmento lleva dirección horizontal y *direc(v)*, que lleva dirección vertical.

```
direc(h).
direc(v).
```

Para representar los segmentos, vamos a considerar el predicado **seg/3**, donde *seg(h,X,Y)* se verifica si se pinta un segmento horizontal que atraviesa las celdas (X,Y) y (X,Y+1), y *seg(v,X,Y)* se verifica si se pinta un segmento vertical que atraviesa las celdas (X,Y) y (X+1,Y).

Generamos así los posibles segmentos que podríamos dibujar en las celdas de la cuadrícula usando una regla de elección:

```
{seg(S,X,Y)} :- direc(S), fila(X), col(Y).
```

Como *seg(S,X,Y)* se verifica si se atraviesa (X,Y) y alguna de las celdas (X+1,Y) o (X,Y+1), según si el segmento es vertical u horizontal respectivamente, en las celdas de la última columna no se podrán realizar segmentos horizontales que partan de ellas, pues no existirán las celdas de la forma (X,m+1), y del mismo modo, tampoco se podrán realizar segmentos verticales que partan de las celdas de la última fila, pues no existirán celdas de la forma (n+1,Y). Representamos esto a través de las siguientes dos reglas:

```
:- seg(h,X,m), fila(X).
:- seg(v,n,Y), col(Y).
```

Esto nos da los posibles conjuntos candidatos a ser solución del puzzle Masyu. Ahora, debemos implementar las reglas, que eliminarán los conjuntos de segmentos que no resuelvan el puzzle.

Primero, representemos que la línea solución del puzzle debe pasar por todos los círculos de la cuadrícula, ya sean blancos o negros. Para representar esto, vamos a usar dos predicados:

1. Usamos el predicado **circulo/2** donde circulo(X,Y) se verifica si la celda (X,Y) contiene un círculo blanco o negro:

```
circulo(X,Y) :- blanco(X,Y), fila(X), col(Y).
circulo(X,Y) :- negro(X,Y), fila(X), col(Y).
```

2. Usamos el predicado **pasaPor/2** donde *pasaPor(X,Y)* representa que alguno de los segmentos que componen la solución pasa por (X,Y).

```
pasaPor(X,Y) :- seg(S,X,Y), fila(X), col(Y), direc(S).
pasaPor(X,Y+1) :- seg(h,X,Y), Y<m, fila(X), col(Y).
pasaPor(X+1,Y) :- seg(v,X,Y), X<n, fila(X), col(Y).
```

Finalmente, se representa que la línea solución debe pasar por todas las celdas que contienen círculos mediante una restricción:

```
:- circulo(X,Y), not pasaPor(X,Y).
```

Implementemos ahora las reglas.

**REGLA 1.**

Debemos representar que la línea solución forma un único camino cerrado que entra y sale de cada celda que atraviesa una única vez.

Para representar que entra y sale de cada celda que atraviesa una única vez, vamos a imponer que la celda sea atravesada únicamente por dos segmentos, ya sea algún segmento que parte de la misma celda en dirección vertical u horizontal u otro segmento que parta de la celda (X-1,Y) en dirección vertical o de (X,Y-1) en dirección horizontal. Por tanto, el cardinal del conjunto formado por estos cuatro segmentos es exactamente 2.

```
:- 3 {seg(h,X,Y); seg(v,X,Y); seg(v,X-1,Y); seg(h,X,Y-1)},
   pasaPor(X,Y).
:- {seg(h,X,Y); seg(v,X,Y); seg(v,X-1,Y); seg(h,X,Y-1)} 1,
   pasaPor(X,Y).
```

Se debe pintar una única línea que sea la solución del puzzle. Esto quiere decir que si la línea solución pasa por dos celdas, podemos llegar de una a otra recorriéndola.

Para representar esto, debemos definir antes el predicado **ady/4**, que representa cuándo dos celdas van a ser adyacentes: una celda va a ser adyacente a (X,Y) si podemos llegar a la celda a través de un segmento que parte de (X,Y) y que forma parte de la línea solución. Así, se define **ady/4** a través de las reglas:

```
ady(X,Y,X+1,Y) :- seg(v,X,Y), X<n.
ady(X,Y,X,Y+1) :- seg(h,X,Y), Y<m.
```

Si (X,Y) es adyacente a una celda (W,Z), la celda (W,Z) será adyacente a (X,Y). Para representar la simetría de la relación, usamos:

```
ady(X,Y,W,Z) :- ady(W,Z,X,Y).
```

Representaremos que si la línea solución atraviesa dos celdas, se puede llegar de una a otra recorriendo la línea usando el predicado **alcanzable/4**, que se define como sigue:

1. Supondremos que toda celda que atraviesa la línea solución es alcanzable desde ella misma.

   ```
   alcanzable(X,Y,X,Y) :- pasaPor(X,Y), fila(X), col(Y).
   ```

2. Supondremos que la celda (W,Z) es alcanzable desde la celda (X,Y) si existe una celda (X1,Y1) de manera que (X,Y) y (X1,Y1) son adyacentes y (W,Z) es alcanzable desde (X1,Y1).

   ```
   alcanzable(X,Y,W,Z) :- ady(X1,Y1,X,Y), alcanzable(X1,Y1,W,Z),
                          fila(X), col(Y), fila(X1), col(Y1),
                          fila(W), col(Z).
   ```

Finalmente, para representar que dada dos celdas por las que la línea solución pasa, siempre debemos llegar de una a otra recorriéndola, usamos una restricción:

```
:- pasaPor(X,Y), pasaPor(W,Z), not alcanzable(X,Y,W,Z).
```

**REGLA 2.**

El camino debe atravesar las celdas con círculos blancos en línea recta, es decir, supongamos que (X,Y) es una celda que contiene un círculo blanco, entonces si de (X,Y) parte un segmento horizontal, a (X,Y) lo debe atravesar otro segmento horizontal, que debe partir de (X,Y-1), y si de (X,Y) parte un segmento vertical, a (X,Y) lo debe atravesar un segmento vertical, esta vez partiendo de (X-1,Y). Para representar esto, se usa una restricción, que impedirá que se dé cualquier otra posibilidad que no sean las que acabamos de exponer:

```
:- 1 {seg(h,X,Y); seg(h,X,Y-1)}, 1 {seg(v,X,Y); seg(v,X-1,Y)},
    blanco(X,Y).
```

Además, la línea solución debe girar 90 grados en la celda anterior o posterior en el camino a la celda que contiene el círculo blanco. Es decir, si la celda (X,Y) contiene un círculo blanco, de ella parte un segmento horizontal y además es atravesada por otro segmento horizontal que parte de (X,Y-1), no puede ocurrir que (X,Y-1) sea atravesada por un segmento horizontal a la vez que de (X,Y+1) parta un segmento horizontal.

```
:- blanco(X,Y), seg(h,X,Y), seg(h,X,Y-1), seg(h,X,Y-2),
    seg(h,X,Y+1).
```

De la misma manera, si la celda (X,Y) que contiene el círculo blanco fuese atravesada por un segmento vertical que parte de (X-1,Y) y además, de ella partiese un segmento vertical, no podría partir a la vez un segmento vertical de (X-2,Y) y de (X+1,Y). Usamos para representarlo la restricción:

```
:- blanco(X,Y), seg(v,X,Y), seg(v,X-1,Y), seg(v,X+1,Y),
    seg(v,X-2,Y).
```

**REGLA 3.**

Por último, vamos a modelizar la regla 3. Primero, representemos que el camino debe realizar giros de 90 grados en las celdas que contengan círculos negros. Para ello, supongamos que (X,Y) es una celda que contiene un círculo negro. Entonces, si de (X,Y) parte un segmento vertical, no puede ocurrir que de (X-1,Y) parta otro segmento vertical, porque si no la celda queda recorrida en línea recta. De igual modo, si de (X,Y) parte un segmento horizontal, no puede ocurrir que de (X,Y-1) parta un segmento horizontal, por el mismo

motivo. Usando restricciones, lo representamos a través de las reglas:

```
:- negro(X,Y), seg(v,X,Y), seg(v,X-1,Y), fila(X), col(Y).
:- negro(X,Y), seg(h,X,Y), seg(h,X,Y-1), fila(X), col(Y).
```

Para representar que las celdas anterior y posterior en el camino a la celda que contiene el círculo negro (que estamos suponiendo que es (X,Y)) deben ser atravesadas en línea recta, se usan cuatro restricciones:

1. Si de la celda (X,Y) parte un segmento horizontal, entonces la celda siguiente en el camino a (X,Y) es (X,Y+1) y para que sea por tanto atravesada el línea recta, de ella no debe partir ni acceder nigún segmento vertical:

   ```
   :- negro(X,Y), seg(h,X,Y), 1 {seg(v,X,Y+1); seg(v,X-1,Y+1)}.
   ```

2. Si de la celda (X,Y-1) parte un segmento horizontal que atraviesa a (X,Y), entonces (X,Y-1) es la celda anterior en el camino a (X,Y) y por tanto, ni de ella ni de (X-1,Y-1) deben partir segmentos verticales para que la línea solución atraviese (X,Y-1) en línea recta.

   ```
   :- negro(X,Y),seg(h,X,Y-1), 1 {seg(v,X,Y-1); seg(v,X-1,Y-1)}.
   ```

3. Si de la celda (X,Y) parte un segmento vertical, la celda siguiente a (X,Y) en el camino es (X+1,Y), luego para que esta se recorra en línea recta, no puede ocurrir que parta de ella un segmento horizontal o parta de (X+1,Y-1) un segmento horizontal.

   ```
   :- negro(X,Y), seg(v,X,Y), 1 {seg(h,X+1,Y); seg(h,X+1,Y-1)}.
   ```

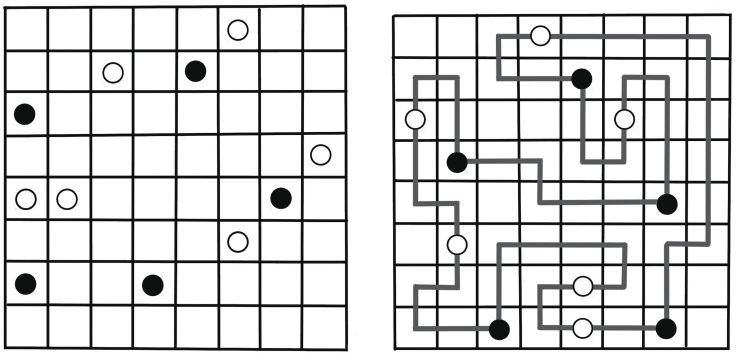
4. Si de la celda (X-1,Y) parte un segmento vertical, esta es la celda anterior a la celda (X,Y) en el camino, luego no puede ocurrir que sea atravesada por un segmento horizontal ni que de ella parta un segmento horizontal.

   ```
   :- negro(X,Y),seg(v,X-1,Y), 1 {seg(h,X-1,Y-1); seg(h,X-1,Y)}.
   ```

La solución del Masyu vendrá dada por los segmentos que finalmente se pinten en la cuadrícula y cumplan las reglas, luego nos fijamos únicamente en los literales formados por el predicado **seg/3** en el conjunto de respuesta:

```
#show seg/3.
```

Como hemos hecho en los puzzles anteriores, vamos a ver algunos ejemplos de puzzle Masyu y sus soluciones.



(a) Masyu nivel normal (b) Solución

Figura 6.5: puzzle Masyu y su solución

**Ejemplo 1.**

Estudiemos primero la solución del puzzle Masyu de la figura 6.5.

Describimos la cuadrícula representando las celdas que poseen un círculo blanco o negro con los predicados **negro/2** y **blanco/2**:

```
blanco(1,4). negro(2,5). blanco(3,1). blanco(3,6). negro(4,2).
negro(5,7). blanco(6,2). blanco(7,5). negro(8,3).
blanco(8,5). negro(8,7).
```

Guardando el código del puzzle Masyu en el archivo `masyu.lp`, la descripción de la cuadrícula en el archivo `masyuNormal.lp` y ejecutando CLINGO de la siguiente manera:

```
clingo masyu.lp masyuNormal.lp -c n=8 -c m=8
```

obtenemos la solución del puzzle:

%seg(h,1,3) seg(h,1,4) seg(h,1,5) seg(h,1,6) seg(h,1,7) seg(h,2,1) seg(h,2,3)
%seg(h,2,4) seg(h,2,6) seg(h,4,2) seg(h,4,3) seg(h,4,5) seg(h,5,1) seg(h,5,4)
%seg(h,5,5) seg(h,5,6) seg(h,6,3) seg(h,6,4) seg(h,6,5) seg(h,6,7) seg(h,7,1)
%seg(h,7,4) seg(h,7,5) seg(h,8,1) seg(h,8,2) seg(h,8,4) seg(h,8,5) seg(h,8,6)
%seg(v,1,3) seg(v,1,8) seg(v,2,1) seg(v,2,2) seg(v,2,5) seg(v,2,6) seg(v,2,7)
%seg(v,2,8) seg(v,3,1) seg(v,3,2) seg(v,3,5) seg(v,3,6) seg(v,3,7) seg(v,3,8)
%seg(v,4,1) seg(v,4,4) seg(v,4,7) seg(v,4,8) seg(v,5,2) seg(v,5,8) seg(v,6,2)
%seg(v,6,3) seg(v,6,6) seg(v,6,7) seg(v,7,1) seg(v,7,3) seg(v,7,4) seg(v,7,7)

**Ejemplo 2.**

Ahora, estudiamos la solución del puzzle de la figura 6.6.

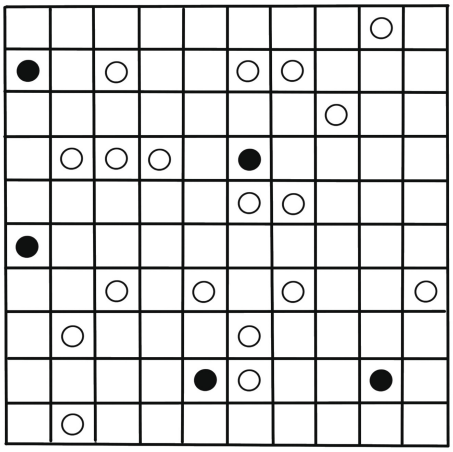


Figura 6.6: puzzle Masyu nivel difícil

Describimos el puzzle a través de las reglas:

```
blanco(1,9). negro(2,1). blanco(2,3). blanco(2,6). blanco(2,7).
blanco(3,8). blanco(4,2). blanco(4,3). blanco(4,4). negro(4,6).
blanco(5,6). blanco(5,7). negro(6,1). blanco(7,3). blanco(7,5).
blanco(7,7). blanco(7,10). blanco(8,2). blanco(8,6). negro(9,5).
blanco(9,6). negro(9,9). blanco(10,2).
```

Guardando esta descripción en `masyuDificil.lp` y ejecutando CLINGO:

```
clingo masyu.lp masyuDificil.lp -c n=10 -c m=10
```

se obtiene la solución siguiente del puzzle:

%seg(h,1,4) seg(h,1,5) seg(h,1,7) seg(h,1,8) seg(h,1,9) seg(h,2,1) seg(h,2,2)
%seg(h,2,3) seg(h,3,2) seg(h,3,4) seg(h,3,7) seg(h,3,8) seg(h,4,6) seg(h,4,7)
%seg(h,4,8) seg(h,5,1) seg(h,5,3) seg(h,5,5) seg(h,5,6) seg(h,5,7) seg(h,5,9)
%seg(h,6,1) seg(h,6,2) seg(h,6,3) seg(h,6,5) seg(h,6,6) seg(h,6,7) seg(h,6,9)
%seg(h,7,2) seg(h,7,3) seg(h,7,6) seg(h,7,7) seg(h,7,8) seg(h,9,2) seg(h,9,3)
%seg(h,9,4) seg(h,9,7) seg(h,9,8) seg(h,10,1) seg(h,10,2) seg(h,10,3)
%seg(h,10,4) seg(h,10,5) seg(h,10,7) seg(h,10,8) seg(h,10,9) seg(v,1,4)
%seg(v,1,6) seg(v,1,7) seg(v,1,10) seg(v,2,1) seg(v,2,6) seg(v,2,7) seg(v,2,10)

%seg(v,3,1) seg(v,3,2) seg(v,3,3) seg(v,3,4) seg(v,3,5) seg(v,3,6) seg(v,3,9)
%seg(v,3,10) seg(v,4,1) seg(v,4,2) seg(v,4,3) seg(v,4,4) seg(v,4,5) seg(v,4,10)
%seg(v,5,8) seg(v,5,9) seg(v,6,1) seg(v,6,4) seg(v,6,5) seg(v,6,10) seg(v,7,1)
%seg(v,7,2) seg(v,7,5) seg(v,7,6) seg(v,7,9) seg(v,7,10) seg(v,8,1) seg(v,8,2)
%seg(v,8,5) seg(v,8,6) seg(v,8,9) seg(v,8,10) seg(v,9,1) seg(v,9,6) seg(v,9,7)
%seg(v,9,10)

# Bibliografía


[CKK+07] Merve Cayli, Ayse Gül Karatop, Ahmet Emrah Kavlak, Hakan Kaynar, Ferhan Ture, and E. Erdem, *Solving challenging grid puzzles with answer set programming*, 2007.

[GK14] Michael Gelfond and Yulia Kahl, *Knowledge representation, reasoning, and the design of intelligent agents*, Cambridge University Press, New York, 2014, The answer-set programming approach.

[GKK+19] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Marius Lindauer, Max Ostrowski, Javier Romero, Torsten Schaub, Sven Thiele, and Philipp Wanko, *Potassco user guide*, 2019.

[GL88] Michael Gelfond and Vladimir Lifschitz, *The stable model semantics for logic programming*, Proceedings of International Logic Programming Conference and Symposium (Robert Kowalski, Bowen, and Kenneth, eds.), MIT Press, 1988, pp. 1070–1080.

[HD21] María José Hidalgo Doblado, *Lógica computacional y teoría de modelos*, `https://www.glc.us.es/~mjoseh/LCyTM2020/index.php/L%C3%B3gica_computacional_y_teor%C3%ADa_de_modelos_(curso_2020-21)`, Curso 2020-2021.

[Lif08] Vladimir Lifschitz, *What is answer set programming?*, vol. 3, 2008, pp. 1594–1597.

[Lif19] Vladimir Lifschitz, *Answer set programming*, Springer International Publishing, 2019.