DCC
DEPARTAMENTO DE CIENCIA DE LA COMPUTACIÓN

Introducción a la Inteligencia Artificial

Apuntes del curso IIC2613

Daniel Florea

PRIMERA EDICIÓN

Índice general

CAPÍTULO 1

Introducción

Nota: Tanto este capítulo como los siguientes del libro seguirán la estructura general de la rendición 2023-1 del curso *IIC2613 - Inteligencia Artificial* de la Pontificia Universidad Católica de Chile.

Si bien puede ser definida de múltiples formas, podemos considerar a la inteligencia artificial como 'métodos, algoritmos y tecnologías que permiten que un software o máquina muestre un comportamiento inteligente, (ojalá) indistinguible al de un humano'. El presente capítulo corresponde a una breve introducción a la historia del concepto y sus distintas áreas de investigación.

1.1 PRIMER ACERCAMIENTO

Pocos años después de crear la primera computadora moderna (la 'Máquina Universal de Turing'), el matemático británico Alan Turing se pregunta a sí mismo '¿pueden las máquinas pensar?'. Dado que *pensar* es un concepto difícil de definir, Turing decide 'reemplazar la pregunta por una nueva, de naturaleza similar pero mucho más concisa'. Es entonces cuando el año 1950 formula lo que hoy en día conocemos como el 'test de Turing'.

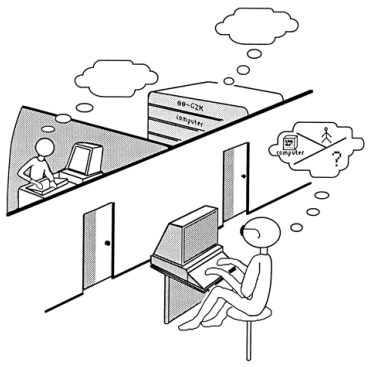

Figura 1.1: Caricatura del test de Turing, Jack Copeland, 2000 [7].

En resumen, el test de Turing (originalmente llamado *imitation game*) busca probar si una máquina 'exhibe comportamiento inteligente' si un evaluador humano no puede distinguir con seguridad a la computadora de una persona al monitorear una conversación escrita entre los dos [12]. Más que importar si las respuestas a las preguntas que se le hacían eran correctas, Turing consideraba más importante **qué tanto se parecían estas a las que un humano otorgaría**, ¿es esto lo que hace a una máquina inteligente?.

1.2 HISTORIA TEMPRANA

Acuñado en 1955, el término *Inteligencia Artificial* fue inventado por el ingeniero en computación John McCarthy y definido como 'la ciencia e ingeniería de hacer máquinas inteligentes'.

Un año después, McCarthy dirige el *'Dartmouth Summer Research Project on Artificial Intelligence'* (DSRPAI), workshop que marca el comienzo de la primera 'primavera de la inteligencia artificial' *(AI Spring)*, término utilizado para indicar periodos de grandes avances en el área [6].

Tras la conferencia de Dartmouth, el campo de la inteligencia artificial vión grandes avances en su desarrollo, entre cuales destacan la creación de uno de los primeros modelos de lenguaje natural, ELIZA (en 1966), capaz de simular conversaciones con otras personas y uno de los primeros programas que mostraron capacidades de poder pasar el test de Turing y el *programa general de resolución de problemas* [8] desarrollada por Herbert Simon, que podía resolver una serie de problemas simples tales como la Torre de Hanoi.

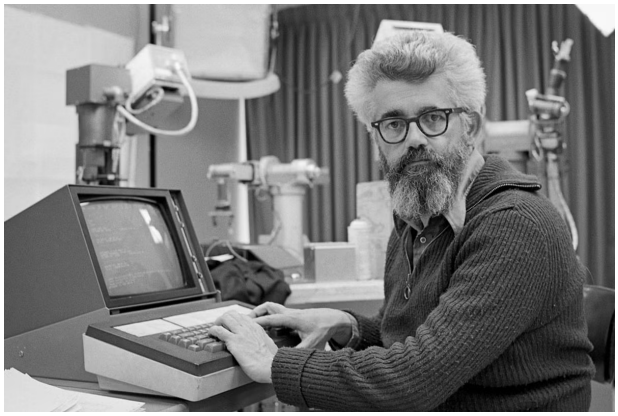

Figura 1.2: John McCarthy en su laboratorio de inteligencia artificial en Stanford, 1974.

Nota: Si deseas probar tener una conversación con el modelo de ELIZA puedes hacerlo en la página `https://www.masswerk.at/elizabot/`.

En base a estos rápidos y notorios avances en el área, el financiamiento para el desarrollo del área es sustancialmente incrementado en 1970, con múltiples expertos afirmando que una máquina con la inteligencia general de una persona humana promedio podría ser desarrollada dentro de tres a ocho años.

Lamentablemente, este no fue el caso y tan solo tres años después, en 1973, el congreso de los Estados Unidos comienza a cuestionar los grandes gastos que se estan haciendo en la investigación de IA. El mismo año, el matemático británico James Lighthill publica un reporte criticando la mirada optimista que los investigadores de IA presentaban, apuntando a que las máquinas solo serían capaces de lograr el nivel de un 'amateur experimentado' en juegos tales como el ajedrez y que habilidades tales como el sentido común se encontraban fuera de su alcance.

En respuesta a las palabras de Lighthill, el Reino Unido decide cancelar el financiamiento para todos los programas de investigación en IA y se entra en el 'invierno de la inteligencia artificial'(*AI Winter*), periodo caracterizado por una baja sustancial en los avances del campo.

1.3 EL RENACIMIENTO DE LA IA

El año 1997, la fabricante de computadores estadounidense IBM desarrolla la supercomputadora *Deep Blue*, diseñada para jugar al ajedrez y la hace competir contra el entonces campeón del mundo Gary Kasparov, venciéndolo en el mejor de 6 partidas por 3½-2½[1] —demostrando como incorrecta una de las observaciones de Lighthill años atrás—reflejando la capacidad de la IA de superar la habilidad humana para determinadas tareas. *Deep Blue* podía procesar 200 millones de movimientos posibles por segundo y determinar el movimiento óptimo a jugar mirando 20 jugadas adelante utilizando una técnica de búsqueda adversaria del tipo alfa beta (ACA VA A IR LA REFERENCIA A ESE CAPÍTULO DE LOS APUNTES CUANDO LO ESCRIBA, POR FAVOR QUE NO SE ME OLVIDE).

[1] No tengo idea de cómo funciona el ajedrez, por lo que no puedo explicar de donde sale el ½

Desde entonces las capacidades de los modelos de inteligencia artificial han crecido exponencialmente; el 2011 el robot de IBM *Watson* gana una competencia del popular juego de televisión *Jeopardy* contra los dos campeones del mundo [5], el 2012 se comienza la revolución de las redes neuronales profundas, el 2016 el campeón del mundo de Go es vencido por la inteligencia *AlphaGo* desarrollada por DeepMind [11] y recientemente, el 2022, el chatbot de OpenAI, *ChatGPT* salió al público y deslumbra por su impresionante habilidad con el lenguaje [2].

1.4 ÁREAS DE INVESTIGACIÓN

Si bien son múltiples las subáreas de trabajo en las que se expande la Inteligencia Artificial, a continuación se destacan las principales dentro del campo:

- **Representación Lógica del Conocimiento y Razonamiento**: Obtener conclusiones y nuevos conocimientos a partir de reglas lógicas.
- **Planificación y Búsqueda Heurística**: Resolución de problemas mediante la búsqueda sobre las acciones posibles.
- **Aprendizaje automático**: Uso de modelos matemáticos para ayudar a un modelo a aprender sin instrucciones directas
- **Visión por Computador**: Extracción de información a partir de imágenes/videos/sensores.
- **Procesamiento de Lenguaje Natural**: Procesamiento y análisis y generación de grandes cantidades de texto, buscando un entendimiento del lenguaje.

Nota: En el presente libro abordaremos solamente problemas de búsqueda, representación lógica, aprendizaje automático y una parte de visión de computador.

Distintos problemas computacionales requieren distintas soluciones pertenecientes a distintas áreas. Es importante entender que la inteligencia artificial no es una herramienta universal, sino más bien el conjunto de herramientas que permiten a una máquina resolver problemas de forma inteligente.

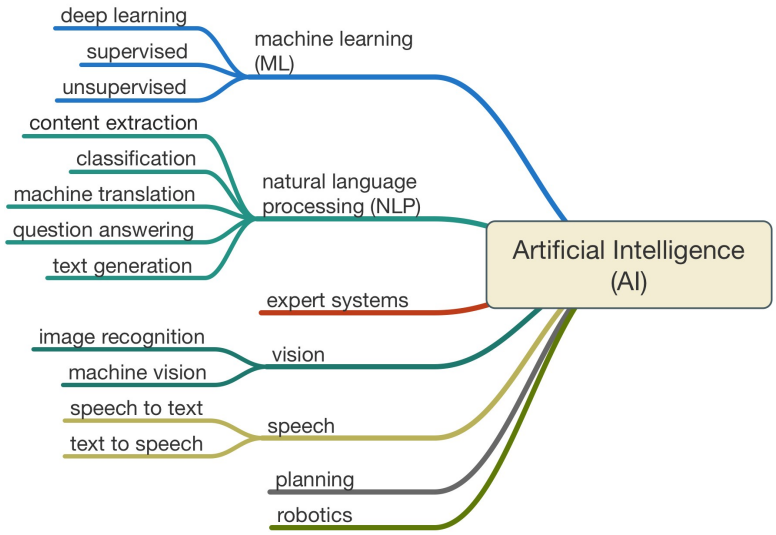

Figura 1.3: Es importante entender a la inteligencia artificial como un área de estudio con múltiples disciplinas, cada una con sus propios algoritmos y modelos.

CAPÍTULO 2

Programación Lógica — ASP

En el siguiente capítulo trabajaremos con un tipo de programación lógica llamada *Answer Set Programmgng* (ASP), un lenguaje declarativo para resolución de problemas de búsqueda difíciles (normalmente NP-hard), bajo una serie de reglas lógicas.

2.1 INSTALACIÓN Y SETUP

Nuestro lenguaje de *Answer Set Programming* a elección para trabajar será clingo [3], el cual nos permite obtener sets de respuestas que representan soluciones a un problema bajo restricciones lógicas escritas bajo la misma sintaxis que un programa de Prolog [1].

A continuación se presentan tutoriales de instalación para clingo en los sistemas operativos de Ubuntu, macOS y Windows, más detalles al respecto se ecuentran en el repositorio oficial del proyecto [9].

Nota: Para quienes no deseen instalar el lenguaje, existe una versión en línea en siguiente link.

2.1.1. Instalación en Ubuntu

1. Clingo puede ser fácilmente instalado en Ubuntu mediante las líneas:

```
$ sudo apt-get update
$ sudo apt-get install gringo
```

2.1.2. Instalación en macOS

1. Para poder instalar clingo en macOS primero deberemos instalar el administrador de paquetes homebrew mediante la siguiente línea en la terminal:

```
$ /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/
    Homebrew/install/HEAD/install.sh)"
```

2. Luego, se deberán instalar las herramientas de la línea de comandos Xcode mediante la siguiente línea (es posible que ya las tengas instaladas):

```
$ xcode-select --install
```

3. Finalmente, clingo puede ser instalado mediante la siguiente línea:

```
$ brew install clingo
```

2.1.3. Instalación en Windows

La instalación en Windows es menos directa que la de otros sistemas operativos, deberemos añadir manualmente el programa al PATH de nuestro equipo.

1. Descarga clingo 5.4.0 desde su página de descargas en GitHub [10] o directamente desde este link.
2. Extrae los componentes del archivo `.zip`, asegúrate de hacerlo en alguna ubicación permanente ya que los archivos deberán permanecer en ella desde ahora en adelante. Luego, copia la dirección en cuál los contenidos fueron extraídos.

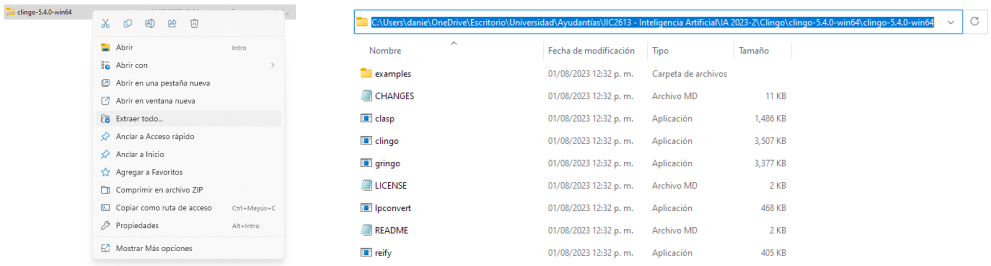

Figura 2.1: Se extraen los contenidos del archivo `.zip` y se copia el directorio.

3. En el buscador de Windows, buscaremos 'Editar las variables de entorno del sistema', abriendo el menú del panel de control y seleccionando el botón en la esquina inferior derecha de 'Variables de entorno...', en este nuevo menú haremos click en PATH y luego en el botón 'Editar...'.

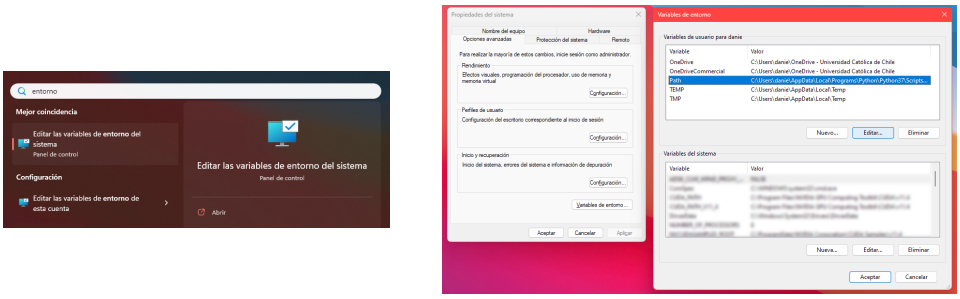

Figura 2.2: Se abre el menú de variables de entorno y se editan las variables en el PATH.

4. Dentro de la pestaña que acaba de abrirse, haremos click en 'Nuevo' para crear un nuevo directorio en el PATH y pegaremos el directorio que copiamos anteriormente (en el que extraímos el `.zip`). Hacemos click en 'Ok' y ahora podremos utilizar clingo correctamente.

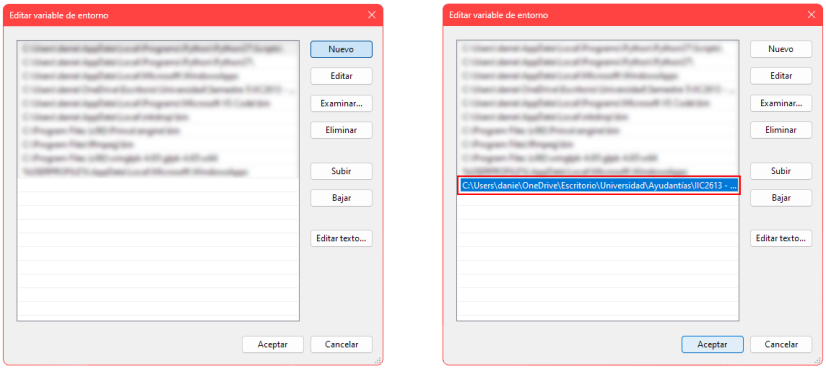

Figura 2.3: Añadimos el directorio de clingo a nuestro PATH.

2.2 USO DE CLINGO

Una vez instalado el lenguaje de programación, podemos empezar a ejecutar programas de programación lógica y a encontrar soluciones para problemas. Clingo hace uso de archivos con extensión `.lp` para su ejecución. Normalmente estos se verán algo así:

graphcoloring.lp

```
% Default
#const n = 3.

% Generate
{ color(X,1..n) } = 1 :- node(X).
% Test
:- edge(X,Y), color(X,C), color(Y,C).

% Nodes
node(1..6).
% (Directed) Edges
edge(1,(2;3;4)).  edge(2,(4;5;6)).  edge(3,(1;4;5)).
edge(4,(1;2)).    edge(5,(3;4;6)).  edge(6,(2;3;5)).

% Display
#show color/2.
```

Podremos ejecutar el archivo con clingo mediante la siguiente línea en la terminal (siempre asegurándonos de que esta se encuentre en la misma carpeta que el archivo):

```
$ clingo graphcoloring.lp
```

Nuestra terminal debería imprimir algo así:

```
clingo version 5.4.0
Reading from graphcoloring.lp
Solving...
Answer: 1
color(2,2) color(1,3) color(3,2) color(4,1) color(5,3) color(6,1)
SATISFIABLE

Models       : 1+
Calls        : 1
Time         : 0.004s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.000s
```

Lo más importante que debemos saber es que el problema es satisfacible (tiene solución) y una de ellas se muestra arriba (después de la línea `Answer: 1`).

Si deseamos obtener **todos** los modelos que solucionan el problema, deberemos ejecutar el programa mediante la línea:

```
$ clingo -n 0 graphcoloring.lp
```

Mediante esta línea, estamos entregándole al programa un atributo `n` que le indica hasta cuantos modelos mostrar, 0 mostrará todos los modelos encontrados, en este caso seis:

```
clingo version 5.4.0
Reading from graphcoloring.lp
Solving...
```

```
Answer: 1
color(2,2) color(1,3) color(3,2) color(4,1) color(5,3) color(6,1)
Answer: 2
color(1,1) color(2,2) color(3,2) color(4,3) color(5,1) color(6,3)
Answer: 3
color(1,1) color(2,3) color(3,3) color(4,2) color(5,1) color(6,2)
Answer: 4
color(1,2) color(2,3) color(3,3) color(4,1) color(5,2) color(6,1)
Answer: 5
color(2,1) color(1,3) color(3,1) color(4,2) color(5,3) color(6,2)
Answer: 6
color(2,1) color(1,2) color(3,1) color(4,3) color(5,2) color(6,3)
SATISFIABLE

Models       : 6
Calls        : 1
Time         : 0.010s (Solving: 0.01s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.000s
```

Por temas de simplicidad, desde ahora en adelante, cada vez que se muestre el retorno de un programa de clingo, se omitirán las líneas sobre la versión, el archivo y el tiempo de cómputo (a menos de que se consideren relevantes).

2.3 INTRODUCCIÓN A CLINGO

Los programas de ASP se contruyen a partir de reglas lógicas. Particularmente, el programa tratará de encontrar todas las soluciones que cumplan la serie de restricciones impuestas por estas reglas. Llamaremos a cada una de estas soluciones que satisfacen el programa como **modelos**.

2.3.1. Reglas Básicas

Definiremos una **regla** en programación en lógica como un objeto de la forma:

$$Head \leftarrow Body$$

Donde *Head* y *Body* son conjuntos de **átomos**. La premisa de toda regla lógica es que si *Body* se encuentra en nuestro modelo (solución), entonces *Head* también debe hacerlo.

En ASP definiremos una regla a partir de variables, por ejemplo:

```
1  p :- q.     % si q se encuentra en el modelo, p también
```

Podemos leer esta regla como 'cada vez que q se encuentra en el modelo, p se encontrará en el modelo'. Ahora bien, si ejecutamos un archivo que solo contenga a esta línea, notaremos que existe una única solución para el programa y corresponde a la respuesta vacía.

```
graphcoloring.lp:1:6-7: info: atom does not occur in any rule head:
  q

Answer: 1

SATISFIABLE

Models       : 1
```

Como señala el mensaje de salida, el átomo q no ocurre en ninguna *Head*, por lo que no existen 'condiciones' para añadirlo al programa.

Una forma de declarar a un átomo como siempre perteneciente al modelo es utilizando un tipo especial de reglas llamadas **hechos**, reglas que solo contienen un *Head*.

```
1  q.          % esto es un hecho
2  p :- q.     % esto es una regla
```

La salida de este programa corresponde a:

```
Answer: 1
q p
SATISFIABLE

Models       : 1
```

Esto es de esperar, ya que declaramos que q se encuentra en el modelo, por lo que p también debe hacerlo según la segunda línea.

Si decidimos introducir un segundo átomo al *Body* de la segunda regla, notamos que este también se debe cumplir para que p pertenezca al modelo:

```
1  q.          % q se encuentra en el modelo
2  r.          % r se encuentra en el modelo
3  p :- q, r.  % si q y r se encuentran en el modelo, p también
```

La salida de este programa corresponde a:

```
Answer: 1
r q p
SATISFIABLE

Models       : 1
```

Ejercicio 2.3.1: Reglas básicas

```
p :- p, q.  % si p y q se encuentran en el modelo, p también
r :- s, t.  % si s y t se encuentran en el modelo, r también
q :- s.     % si s se encuentra en el modelo, q también
p.          % p se encuentra en el modelo
```

¿Qué modelo es la única solución a este programa??

a) {p}
b) {p, q}
c) Vacío
d) El problema es insatisfacible (no existe un modelo que lo solucione)

Solución 2.3.1

Notamos que p es declarado como parte del modelo en la cuarta línea, sabiendo esto, notamos que su presencia no significa el cumplimiento de ninguna otra lógica, por lo que no se añaden más átomos al modelo.

De este modo, el modelo generado por el programa sería a), {p}

No todas las reglas de un átomo en su head deben cumplirse para que este se encuentre dentro de un modelo solución (cada regla para un mismo head actúa como una disyunción, es decir, como un OR de las otras reglas):

```
q.          % q se encuentra en el modelo
p :- q, r.  % si q y r se encuentran en el modelo, p también
p :- q.     % si q se encuentra en el modelo, p también
```

La salida de este programa corresponde a:[1]

```
Answer: 1
q p
SATISFIABLE

Models       : 1
```

Ejercicio 2.3.2: Dependencias circulares

¿Qué ocurre cuando un programa tiene dependencias circulares en sus reglas?

```
q.          % q se encuentra en el modelo
p :- q, r.  % si q y r se encuentran en el modelo, p también
r :- p.     % si p se encuentra en el modelo, r también
```

Solución 2.3.2

Cuando un programa tiene dependencias circulares, las reglas no se cumplirán a menos de que exista otro conjunto de reglas que si incluyan a su *Body* en el modelo.

El código del ejercicio nos entregaría un único modelo de solución que contiene solamente a `q`.

Una variante del programa que si nos entregaría un modelo con las tres variables sería:

```
q.          % q se encuentra en el modelo
p :- q, r.  % si q y r se encuentran en el modelo, p también
r.          % r se encuentra en el modelo
r :- p.     % si p se encuentra en el modelo, r también
```

Notamos que el orden de las reglas no es relevante para el cálculo del modelo.

Por último, podemos generar un hecho que especifique que un átomo no puede estar en el modelo solución mediante:

```
:- p.   % p no formará parte del modelo
```

Ejercicio 2.3.3: Contradicción entre dos hechos

¿Cuál es la salida del siguiente programa?

```
p.
:- p.
```

[1] Se ignora el hecho de que `r` no se encuentre en ninguna regla como *Head* para mantener la simpleza del ejemplo

Solución 2.3.3

Dado que ambas reglas presentan una contradicción lógica, no existen modelos que solucionen el problema, por tanto es considerado INSATISFACIBLE.

Si ejecutamos el programa en nuestra terminal, obtendremos una salida así:

```
Solving...
UNSATISFIABLE

Models       : 0
```

2.3.2. Variables y Proposiciones

Definiremos una **proposición** como una propiedad que se puede cumplir o no por un término definido por el usuario y que puede ser verdadera o falsa. Una proposición estará compuesta por un predicado y un término.

```
1  profesor(jorge).    % el término 'jorge' cumple el predicado 'profesor'
2  profesor(hans).     % el término 'hans' cumple el predicado 'profesor'
3  ayudante(daniel).   % el término 'daniel' cumple el predicado 'ayudante'
4  estudiante(ria).    % el término 'ria' cumple el predicado 'estudiante'
```

Nota: Los términos también pueden ser escritos como *strings*, por lo tanto, utilizar profesor('jorge') generaría un término 'jorge' en lugar de `jorge`.

La salida de este programa corresponde a:

```
Answer: 1
profesor(jorge) profesor(hans) ayudante(daniel) estudiante(ria)
SATISFIABLE

Models       : 1
```

Una forma de generar nuevos predicados a partir de aquellos declarados como hechos es utilizando **variables**, estas nos permiten trabajar con términos genéricos y definir nuevos predicados a partir de ellos. La distinción entre una variable y un término radica en el uso de mayúsculas o minúsculas respectivamente.

De este modo, en lugar de tener que escribir

```
1  profesor(jorge).    % el término 'jorge' cumple el predicado 'profesor'
2  profesor(hans).     % el término 'hans' cumple el predicado 'profesor'
3  ayudante(daniel).   % el término 'daniel' cumple el predicado 'ayudante'
4  estudiante(ria).    % el término 'ria' cumple el predicado 'estudiante'
5
6  curso(jorge) :- profesor(jorge).    % si 'jorge' es profesor, forma parte del curso
7  curso(hans) :- profesor(hans).      % si 'hans' es profesor, forma parte del curso
8  curso(daniel) :- ayudante(daniel).  % si 'daniel' es ayudante, forma parte del curso
9  curso(ria) :- estudiante(ria).      % si 'ria' es estudiante, forma parte del curso
```

Podemos escribir el código equivalente

```
profesor(jorge).    % el término 'jorge' cumple el predicado 'profesor'
profesor(hans).     % el término 'hans' cumple el predicado 'profesor'
ayudante(daniel).   % el término 'daniel' cumple el predicado 'ayudante'
estudiante(ria).    % el término 'ria' cumple el predicado 'estudiante'

curso(X) :- profesor(X).    % todo profesor forma parte del curso
curso(X) :- ayudante(X).    % todo ayudante forma parte del curso
curso(X) :- estudiante(X).  % todo estudiante forma parte del curso
```

Notamos que las salidas de ambos códigos son las mismas:

```
Answer: 1
profesor(jorge) profesor(hans) curso(ria) curso(daniel) curso(jorge) curso(hans) ayudante(daniel)
    estudiante(ria)
SATISFIABLE

Models       : 1
```

Notamos que rápidamente el número de predicados en nuestro modelo comienza a crecer. Muchas veces esto puede resultar engorroso y poco práctico a la hora de leer la salida de un programa, es aquí donde Clingo nos permite utilizar los denominados *statements* (o en español, declaraciones). Características cuyo rol va más alla de los contenidos de un programa [4].

Particularmente, nos interesa el *statement* **`#show predicado/N.`**.

Uso de #show:
`#show` nos permite indicarle a Clingo que solamente queremos visualizar instancias de un determinado predicado (o una serie de predicados) en nuestro programa, por ejemplo, en el código anterior podríamos agregar una línea al final que indique que solamente queremos ver si alguien forma parte de un curso o no mediante el *statement* '`#show curso/1.`'.

De este modo, el código

```
profesor(jorge).    % el término 'jorge' cumple el predicado 'profesor'
profesor(hans).     % el término 'hans' cumple el predicado 'profesor'
ayudante(daniel).   % el término 'daniel' cumple el predicado 'ayudante'
estudiante(ria).    % el término 'ria' cumple el predicado 'estudiante'

curso(X) :- profesor(X).    % todo profesor forma parte del curso
curso(X) :- ayudante(X).    % todo ayudante forma parte del curso
curso(X) :- estudiante(X).  % todo estudiante forma parte del curso

#show curso/1.
```

Tendrá como salida en terminal:

```
Answer: 1
curso(ria) curso(daniel) curso(jorge) curso(hans)
SATISFIABLE

Models       : 1
```

Una característica relevante sobre Clingo es que **un mismo predicado puede estar asociado a múltiples proposiciones**, es aquí donde el valor de N que utilizemos para `#show predicado/N.` gana importancia.

Supongamos que tenemos un predicado `catedra` que indica si un docente puede hacer una clase. Los profesores pueden hacer clases de forma individual o en parejas, por tanto, definiremos dos proposiciones para este predicado:

```
profesor(jorge).    % el término 'jorge' cumple el predicado 'profesor'
profesor(hans).     % el término 'hans' cumple el predicado 'profesor'
ayudante(daniel).   % el término 'daniel' cumple el predicado 'ayudante'
estudiante(ria).    % el término 'ria' cumple el predicado 'estudiante'

catedra(X) :- profesor(X).                  % Solo profesores pueden hacer cátedras
catedra(X, Y) :- profesor(X), profesor(Y).

#show catedra/1.    % Visualizamos las cátedras con un profesor
#show catedra/2.    % Visualizamos las cátedras con dos profesores
```

Si bien `#show` funciona correctamente, notamos que hay algo extraño respecto a la salida del programa...

```
Answer: 1
catedra(jorge) catedra(hans) catedra(jorge,jorge) catedra(hans,jorge) catedra(jorge,hans) catedra(
    hans,hans)
SATISFIABLE

Models       : 1
```

¡Los profesores están haciendo cátedras en parejas consigo mismos! A menos de que se trate de la mismísima oveja Dolly que esté haciendo clases[2], esto no debería ocurrir. Afortunadamente, Clingo nos permite usar operadores entre variables para poder expresar condiciones lógicas más complejas. En este caso, queremos que las variables `X` e `Y` sean distintas entre sí[3], por lo que deberemos reescribir el código como:

```
profesor(jorge).    % el término 'jorge' cumple el predicado 'profesor'
profesor(hans).     % el término 'hans' cumple el predicado 'profesor'
ayudante(daniel).   % el término 'daniel' cumple el predicado 'ayudante'
estudiante(ria).    % el término 'ria' cumple el predicado 'estudiante'

catedra(X) :- profesor(X).                  % Solo profesores pueden hacer cátedras
catedra(X, Y) :- profesor(X), profesor(Y), X != Y.  % X debe ser distinto a Y

#show catedra/1.    % Visualizamos las cátedras con un profesor
#show catedra/2.    % Visualizamos las cátedras con dos profesores
```

La salida del programa ya no cometerá el error anteriormente mencionado:

```
Answer: 1
catedra(jorge) catedra(hans) catedra(hans,jorge) catedra(jorge,hans)
SATISFIABLE

Models       : 1
```

Por último, notamos de que el orden en cual se presentan los profesores en la cátedra no nos importa realmente. Ahora bien, si nos interesa que solamente uno de estos se muestre en la solución.

[2] `https://es.wikipedia.org/wiki/Oveja_Dolly`
[3] Para una lista en detalle de los comparadores que podemos utilizar, se recomienda revisar la guía oficial de POTASSCO [4]

¿Existirá alguna forma de obtener una (o más de una) respuesta que solamente contenga una de las combinaciones entre los términos?

2.4 PROPIEDADES AVANZADAS DE CLINGO

2.4.1. Negación en Clingo

Al final de 2.3.1 definimos cómo enunciar que un átomo se encuentra fuera del modelo. Naturalmente, uno podría pensar de que esto se trata de negación en Clingo, aún cuando este no es el caso.

La negación en Clingo se lleva a cabo mediante la palabra **`not`**, `not` nos indica la ausencia de un átomo (o proposición) en el modelo.[4]

La mejor forma de ilustrar su funcionamiento es mediante un ejemplo:

Ejercicio 2.4.1: Negación básica

¿Son equivalentes los siguientes programas?

```
p :- q. % p está en el modelo si q está
:- q.   % q no está en el modelo
```

```
p :- not q. % p está en el modelo si
            % este no contiene a q
```

Solución 2.4.1

Si bien ambas líneas de código pueden verse muy similares entre sí, su principal diferencia se hace evidente cuando estudiamos las salidas obtenidas al ejecutarlos:

```
Answer: 1

SATISFIABLE

Models       : 1
```

```
Answer: 1
p
SATISFIABLE

Models       : 1
```

Mientras que el primer código no incluye a `p` al modelo dado que `q` no se encuentra en este, el segundo **si lo hace exactamente porque `q` no pertenece al modelo.**

Por tanto, `not` nos permite crear reglas lógicas que indiquen la necesidad de que un átomo se encuentre fuera del modelo para ser verdaderas. Ahora bien, esto no quiere decir que estemos sacando al átomo en cuestión de todo modelo solución, un ejemplo que ilustra esto se presenta a continuación:

Ejercicio 2.4.2: Negación básica parte 2

```
q.          % q está en el modelo
p :- not q. % p está en el modelo si
            % este no contiene a q
```

¿Cuál es el modelo generado por el siguiente programa?

a) {p}
b) {q}
c) El problema es insatisfacible (no existe un modelo que lo solucione)

[4] Desde ahora en adelante, cuando se indique que algo se cumple para átomos, se puede asumir que también lo hará para proposiciones a menos de que se mencione lo contrario.

Solución 2.4.2

Los comentarios del código nos ayudan bastante a entender la lógica detrás de la ejecución del programa, dado que `q` se encuentra en el modelo, sabemos que `p` no se encuentra dentro de este (`p` solo pertenece al modelo si `q` no lo hace).

Si ejecutamos el programa obtendremos la siguiente salida:

```
Answer: 1
q
SATISFIABLE

Models       : 1
```

Por lo que la respuesta correcta sería la alternativa b).

Ahora que entendemos un poco mejor de como funciona la negación en Clingo podemos volver a la pregunta final que nos hicimos en 2.3.2, *¿existirá alguna forma de obtener una (o más de una) respuesta que solamente contenga una de las combinaciones entre los términos?*

La respuesta es sí, y podemos obtenerla añadiendo una sola negación a una de las reglas del programa:

```
1  profesor(jorge).    % el término 'jorge' cumple el predicado 'profesor'
2  profesor(hans).     % el término 'hans' cumple el predicado 'profesor'
3  ayudante(daniel).   % el término 'daniel' cumple el predicado 'ayudante'
4  estudiante(ria).    % el término 'ria' cumple el predicado 'estudiante'
5
6  catedra(X) :- profesor(X).                 % Solo profesores pueden hacer cátedras
7
8  % Añadimos la negación 'not catedra(Y, X)'
9  % asi solo existirá UNA combinación entre X e Y en cada modelo
10 catedra(X, Y) :- profesor(X), profesor(Y), X != Y, not catedra(Y, X).
11
12 #show catedra/1.    % Visualizamos las cátedras con un profesor
13 #show catedra/2.    % Visualizamos las cátedras con dos profesores
```

La salida del programa tras añadir la condición anterior será:[5]

```
Answer: 1
catedra(jorge) catedra(hans) catedra(hans,jorge)
Answer: 2
catedra(jorge) catedra(hans) catedra(jorge,hans)
SATISFIABLE

Models       : 2
```

Notamos que ahora existen múltiples modelos que satisfacen al programa (el programa tiene más de una solución); una que contiene la combinación `catedra(hans, jorge)` y otra que contiene la combinación `catedra(jorge, hans)`. Esto se debe a que la negación hace que la presencia de una en el modelo sea mutuamente excluyente a la otra, por lo que existen dos combinaciones posibles de satisfacer esa regla.

[5] Recuerda que para obtener todos los modelos solución de un programa este debe ser ejecutado con `clingo -n 0 archivo.lp`, más información al respecto se comenta al final de la sección 2.2

Ejercicio 2.4.3: Negaciones circulares

```
q :- not p. % q está en el modelo si
            % este no contiene a p
p :- not q. % p está en el modelo si
            % este no contiene a q
```

¿Cuál es el(los) modelo(s) generado(s) por el siguiente programa?

a) {p}
b) {q}
c) {q}, {p}
d) El problema es insatisfacible (no existe un modelo que lo solucione)
e) El modelo vacío ({})

Solución 2.4.3

Por el funcionamiento de `not`, sabemos que q está en todo modelo que no contiene a p y p en todo modelo que no contiene a q. Por tanto, existen dos modelos que satisfacen el problema, los cuales pueden ser visualizados al ejecutar el código:

```
Answer: 1
p
Answer: 2
q
SATISFIABLE

Models       : 2
```

Por lo que la respuesta correcta sería la alternativa c).

Una forma de pensar de entender la resolución del programa es con prueba y error utilizando todos los modelos posibles, en este caso serían 4:

- El modelo vacío { }:
  Este modelo no satisface la primera regla del programa, dado que p no forma parte de él, q si debería hacerlo.
- **{q}:**
  Este modelo si satisface el programa, esto debido a que la segunda regla no se cumple, por lo que p no forma parte del modelo (y por ende q si).
- **{p}:**
  Este modelo si satisface el programa, esto debido a que la primera regla no se cumple, por lo que q no forma parte del modelo (y por ende p si).
- {p}, {q}:
  Este modelo no satisface el programa, debido a que la primera regla no se cumple (la segunda tampoco), esto debido a que p forma parte del modelo, por lo que q no debería formar parte.

Si bien la forma en que clingo optimiza un programa no es mediante la prueba y error de todas las soluciones posibles, si es una manera efectiva de entender el funcionamiento de un programa. Para más detalles sobre cómo Clingo encuentra modelos soluciones para un problema se recomienda visitar la guía oficial de su funcionamiento. [4]

Por último, podemos omitir la presencia de algún átomo en el modelo solución creando un hecho con la negación de este, por tanto, una línea como:

```
1  not q.
```

Significará que ningún modelo solución contendrá a q. Esto gana especial en el siguiente apartado cuando estudiemos Restricciones de Cardinalidad en Clingo (2.4.2).

Ejercicio 2.4.4: Negación de una proposición

```
profesor(jorge).    % el término 'jorge' cumple el predicado 'profesor'
profesor(hans).     % el término 'hans' cumple el predicado 'profesor'
ayudante(daniel).   % el término 'daniel' cumple el predicado 'ayudante'
estudiante(ria).    % el término 'ria' cumple el predicado 'estudiante'

catedra(X) :- profesor(X).
catedra(X, Y) :- profesor(X), profesor(Y), X != Y, not catedra(Y, X).

not catedra(hans, jorge).

#show catedra/1.
#show catedra/2.
```

¿Cuál es el(los) modelo(s) generado(s) por el siguiente programa (solo lo visualizado)?

a) {catedra(jorge) catedra(hans) catedra(jorge, hans)}
b) {catedra(jorge) catedra(hans) catedra(hans, jorge)}
c) El modelo vacío ({})
d) El problema es insatisfacible (no existe un modelo que lo solucione)
e) {catedra(jorge) catedra(hans) catedra(hans, jorge)},
   {catedra(jorge) catedra(hans) catedra(jorge, hans)}

Solución 2.4.4

Dado que declaramos que `catedra(hans, jorge)` no puede pertenecer al modelo, tenemos que todo modelo que contenga aquella solución ahora no satisface al problema, por lo que ahora existe solamente una única forma de solucionar el problema (la que contiene a `catedra(jorge, hans)` en lugar de `catedra(hans, jorge)`):

```
Answer: 1
catedra(jorge) catedra(hans) catedra(jorge,hans)
SATISFIABLE

Models       : 1
```

Por lo que la respuesta correcta sería la alternativa a).

Notamos que el uso de `not` en el head nos permite omitir algunas respuestas de los modelos solución en base a cierto criterio, esto gana especial relevancia a la hora de modelar un problema en Clingo, (como haremos en la sección 2.5).

2.4.2. Restricciones de Cardinalidad en Clingo

Las restricciones de cardinalidad en Clingo nos permiten restringir el número de átomos en el head que añadiremos al modelo para un mismo body. Esto resulta en la generación de múltiples modelos dependiendo de cuáles head fueron seleccionados.

Un simple ejemplo para esto se encuentra a continuación:[6]

```
amo.                          % amo se encuentra en el modelo
{gato; perro; loro} :- amo.   % añadiremos cero o más de los elementos
                              % por modelo si 'amo' se encuentra en el modelo
```

La salida de este programa corresponde a:

```
Answer: 1
amo
Answer: 2
amo perro
Answer: 3
amo loro
Answer: 4
amo perro loro
Answer: 5
amo gato
Answer: 6
amo gato loro
Answer: 7
amo gato perro
Answer: 8
amo gato perro loro
SATISFIABLE

Models       : 8
```

Por defecto, la restricción de cardinalidad va a probar todas las combinaciones posibles entre los elementos de los paréntesis, podemos restringir esto mediante el uso de números a ambos lados:

```
amo.                              % amo se encuentra en el modelo
1 {gato; perro; loro} 2 :- amo.   % añadiremos al menos 1 y hasta 2 átomos
                                  % por modelo si 'amo' se encuentra en este
```

```
Answer: 1
amo perro
Answer: 2
amo gato
Answer: 3
amo perro gato
Answer: 4
amo loro
Answer: 5
amo loro gato
Answer: 6
amo loro perro
SATISFIABLE

Models       : 6
```

[6] Se usa la palabra 'amo' debido a que clingo no acepta la letra ñ dentro de sus programas, por lo que 'dueño' no era posible.

Ejercicio 2.4.5: Interacción entre una restricción y otras reglas lógicas

El siguiente programa retorna un solo modelo, dado que declaramos la inhabilidad de `catedra(hans, jorge)` de formar parte de una solución:

```
profesor(jorge).    % el término 'jorge' cumple el predicado 'profesor'
profesor(hans).     % el término 'hans' cumple el predicado 'profesor'

catedra(X, Y) :- profesor(X), profesor(Y), X != Y, not catedra(Y, X).
not catedra(hans, jorge).

#show catedra/2.
```

¿Qué ocurre con su salida si añadimos la regla con restricción de cardinalidad:

```
1 {catedra(hans, jorge); catedra(jorge, hans)} 2.
```

dentro de este?

Solución 2.4.5

Las restricciones de cardinalidad trabajan bajo las restricciones del resto de las reglas en el programa, por tanto, la línea que prohíbe a `catedra(hans, jorge)` de formar parte de un modelo solución, evita que esa combinación de la restricción sea 'extraída' y forme parte de un modelo.[a]

```
Answer: 1
catedra(jorge) catedra(hans) catedra(jorge,hans)
SATISFIABLE

Models       : 1
```

Recordemos que es útil pensar que Clingo 'prueba' soluciones para sus problemas y se queda con aquellas que si cumplen todas las reglas lógicas dentro de este.

[a] En realidad si se prueba con aquella combinación, pero al no cumplir con las reglas lógicas del programa, se descarta y no se muestra en los modelos solución de la salida.

Volviendo al código anterior, imaginemos que ahora queremos tener múltiples dueños para distintos animales, un primer acercamiento para la tarea sería un código como el siguiente:

```
1 amo(vicente).   % Vicente es amo de mascotas
2 amo(ignacio).   % Ignacio es amo de mascotas
3 1 {gato; perro; loro} 2 :- amo(X).
```

Si estudiamos la salida del programa notamos que no existe una asociación clara entre cada mascota y su amo:

```
Answer: 1
amo(vicente) amo(ignacio) perro
Answer: 2
amo(vicente) amo(ignacio) gato
Answer: 3
amo(vicente) amo(ignacio) perro gato
Answer: 4
amo(vicente) amo(ignacio) loro
```

```
Answer: 5
amo(vicente) amo(ignacio) loro gato
Answer: 6
amo(vicente) amo(ignacio) loro perro
SATISFIABLE

Models       : 6
```

Una forma de arreglar este problema es generando una proposición para representar esta asociación, como se muestra a continuación:

```
amo(vicente).   % Vicente es amo de mascotas
amo(ignacio).   % Ignacio es amo de mascotas
1 {
 amo_de(X, gato),
 amo_de(X, perro),
 amo_de(X, loro)
} 2 :- amo(X).    % Definimos la proposición amo_de
```

Ahora tenemos una clara asociación entre cada mascota y su amo, pero podemos simplificar el código aún más, generalizando para cualquier mascota, sin necesidad de especificarlas en la reducción:

```
amo(vicente).   % Vicente es amo de mascotas
amo(ignacio).   % Ignacio es amo de mascotas

mascota(gato).  % Un gato es una mascota
mascota(perro). % Un perro es una mascota
mascota(loro).  % Un loro es una mascota

1 { amo_de(X, A): mascota(A) } 2 :- amo(X). % Algunos animales tienen amo
```

Este problema tiene 36 soluciones distintas, por lo que solo se mostrarán un par de ellas:

```
Answer: 1
amo_de(vicente,perro) amo_de(ignacio,gato)
Answer: 2
amo_de(vicente,loro) amo_de(ignacio,gato)
(...)
Answer: 6
amo_de(vicente,perro) amo_de(ignacio,perro)
(...)
Answer: 36
amo_de(vicente,gato) amo_de(vicente,perro) amo_de(ignacio,gato) amo_de(ignacio,perro)
SATISFIABLE

Models       : 36
```

Notamos que un distintos dueños pueden tener una misma mascota, en caso de desear lo contrario deberemos añadir una restricción en el body de la regla.

```
amo(vicente).   % Vicente es amo de mascotas
amo(ignacio).   % Ignacio es amo de mascotas

mascota(gato).  % Un gato es una mascota
mascota(perro). % Un perro es una mascota
```

```
mascota(loro).  % Un loro es una mascota

% (Esto es una sola regla con restricción de cardinalidad)
% Se puede ser amo de una mascota si ningún otro amo distinto está asociado de ella
1 {
 amo_de(X, A): mascota(A)
} 2 :- amo(X), not amo_de(Y, A), amo(Y), mascota(A), X != Y.

#show amo_de/2.
```

El cual nos generaría solamente 8 modelos.

Ejercicio 2.4.6: Modelación básica con restricción de cardinalidad

¿Cómo modelamos todos los equipos posibles a crear entre 8 tenistas que jugarán partidas de dobles? El número del equipo es relevante, por lo que una misma pareja en equipos de distinto número son soluciones distintas.

Solución 2.4.6

tennis_teams.lp

```
% Definimos a los jugadores
jugador("Tenista 1").
jugador("Tenista 2").
jugador("Tenista 3").
jugador("Tenista 4").
jugador("Tenista 5").
jugador("Tenista 6").
jugador("Tenista 7").
jugador("Tenista 8").

equipo(1..4).   % Otra forma de decir equipo(1). equipo(2). ...

% Un jugador no puede estar en dos equipos.
% No se pueden repetir jugadores en un equipo.

% Todos los jugadores pertenecen a un equipo.
1 { juega_en(J, E) : equipo(E) } 1 :- jugador(J).

% Todos los equipos tienen 2 jugadores.
2 { juega_en(J, E) : jugador(J) } 2 :- equipo(E).

#show juega_en/2.
```

El código genera las 2520 combinaciones posibles a llevar a cabo.

Usando restricciones de cardinalidad podemos limitar la cantidad de veces que una proposición ocurre en un programa, no solo para todas sus instancias, si no para un cierto valor dentro de la preposición (por ejemplo, cuando limitamos el número de jugadores por cada equipo).

2.4.3. Maximización/Minimización de problemas en Clingo

2.5 MODELACIÓN DE PROBLEMAS EN CLINGO

Bibliografía

[1] Dic. de 2022. URL: https://es.wikipedia.org/wiki/Prolog.

[2] URL: https://chat.openai.com/chat.

[3] URL: https://potassco.org/clingo/.

[4] URL: https://github.com/potassco/guide/releases/.

[5] David A Ferrucci. "Introduction to "this is watson"". En: *IBM Journal of Research and Development* 56.3.4 (2012), págs. 1-1.

[6] Michael Haenlein y Andreas Kaplan. "A Brief History of Artificial Intelligence: On the Past, Present, and Future of Artificial Intelligence". En: *California Management Review* 61.4 (2019), págs. 5-14. DOI: 10.1177/0008125619864925. eprint: https://doi.org/10.1177/0008125619864925. URL: https://doi.org/10.1177/0008125619864925.

[7] Jack Copeland. *The Turing Test*. [Online; accedido el 31 de Julio de 2023]. 2000. URL: https://www.alanturing.net/turing_archive/pages/reference%20articles/theturingtest.html.

[8] Allen Newell, John C Shaw y Herbert A Simon. "Report on a general problem solving program". En: *IFIP congress*. Vol. 256. Pittsburgh, PA. 1959, pág. 64.

[9] Potassco. *POTASSCO/Clingo: a grounder and solver for logic programs*. URL: https://github.com/potassco/clingo/tree/master.

[10] Potassco. *Releases · POTASSCO/Clingo*. URL: https://github.com/potassco/clingo/releases.

[11] David Silver y col. "Mastering the game of Go with deep neural networks and tree search". En: *nature* 529.7587 (2016), págs. 484-489.

[12] Alan M Turing. "1. THE IMITATION GAME". En: *Theories of Mind: An Introductory Reader* (2006), pág. 51.