Resumen: Razonamiento Lógico

Carro Barallobre, Icia

May 26, 2020

Contents

1 Introducción

Históricamente, uno de los primeros desafíos para la representación del conocimiento ha sido el razonamiento sobre acciones y cambios. En 1969, McCarthy & Hayes introdujeron "Situation Calculus" basado en lógica de primer orden, fluentes, acciones y situaciones. Los problemas de razonamiento más típicos son: Simulaciones, explicaciones temporales, planificación, diagnósticos y verificaciones.

La representación del conocimiento debe ser simple, de semántica clara y permitir la automatización del razonamiento. Una forma de representar el conocimiento es la lógica clásica pero tiene algunos problemas:

- **Frame problem**: Añadir un fluente hace que tengamos que reformular completamente el problema. La lógica clásica no contempla el razonamiento por defecto ya que tiene una relación de inferencia monótona: Siendo Γ la base del conocimiento (KB) y α las conclusiones que se derivan de Γ ($\Gamma \vDash \alpha$), si a Γ le añadimos Δ, las conclusiones anteriores deben mantenerse ($\Gamma \cup \Delta \vDash \alpha$).
- **Qualification problem**: Las precondiciones son afectadas por las condiciones que habilitan la acción. Por lo que, por ejemplo, cada vez que haya una excepción se debe reescribir otra vez todo.
- **Ramification problem**: Arrastrar efectos indirectos, produce ramificaciones.

La solución a estos problemas es el razonamiento por defecto, en el que en ausencia de información saltamos a la conclusión y, el razonamiento no monótono ($\Gamma \vDash \alpha$, pero $\Gamma \cup \Delta \nvDash \alpha$).

2 Representación proposicional y razonamiento

2.1 Lógica proposicional

Para razonar usando lógica proposicional debemos llegar desde una serie de premisas a una conclusión $KB = \{P_1, ..., P_n\} \vDash C$. Si todas las premisas KB son modelos de la conclusión C, es decir, si todas las premisas son ciertas, diremos que C es una consecuencia semántica de KB. Podemos corroborar si C es conclusión de KB de las siguientes formas:

- $P_1 \wedge P_2 \wedge ... \wedge P_n \rightarrow C$ es una tautología ($\top$) como comentamos anteriormente.
- $P_1 \wedge ... \wedge P_n \wedge \neg C$ es una inconsistencia ($\bot$) $*^2$

2.2 SAT

El problema de decidir si usando lógica proposicional se tiene o no algún modelo se denomina **SAT**. Es un problema de decisión, es decir, dada una entrada se nos responde sí o no hay algun modelo. Este problema es **NP-completa**, ya que en el peor de los casos esa es su complejidad.

Este tipo de problemas se pueden resolver mediante:

- **Tablas de verdad**: Chequear las (2^n) posibilidades siendo n el n^o de premisas para ver hay algún caso. Es una solución sencilla de implementar pero, computacionalmente, mala: siempre estás en el peor de los casos con coste 2^n. ($*^2$)
- **Resolution rule**: Este algoritmo usa reglas de inferencia básicas que se aplican sobre cláusulas CNF. Estas reglas se basan en que si tenemos $\alpha \vee p$ y $\beta \vee \neg p$ entonces derivaremos $\alpha \vee \beta$, así obtenemos más información con objetivo de llegar a un punto donde tengamos p y $\neg p$: lo que es una contradicción y hace que el problema sea insatisfactible. ¿Qué problema tiene esta solución? Cada vez tenemos más cláusulas (2^{n+1} de cada vez) y sin una heurística que nos guíe puede ser una tarea complicada. Además, pasar a CNF estando en DNF puede tener coste exponencial.

2.3 Claúsulas de Horn y programas lógicos positivos

Para poder aplicar **resolution rule** de forma eficiente, las fórmulas deben tener una forma particular.

En esta asignatura hemos dado las claúsulas de Horn que, aunque no todos los problemas se puedan representar de esta forma, hay un subconjunto bastante amplio de ellos que si encajan. Si un problema CNF contiene exclusivamente clausulas de Horn, la complejidad del problema SAT (**HORNSAT**) pasa a ser **P-completo**.

Una **cláusula de Horn** es aquella que contiene como mucho un literal positivo. Por lo que tenemos 3 posibles casos:

- Hecho - Un literal positivo.

  $$head\ (n = 0).$$

- Regla - Literal positivo y el resto negativos.

  $$head \vee \neg body_1 \vee ... \vee \neg body_n \equiv head \leftarrow body_1 \wedge ... \wedge body_n,\ (n > 0).$$

  $$head : -body_1, ..., body_n,\ (n > 0).$$

- Prohibición: Todos los literales negativos.

  $$\neg p_1 \vee ... \vee \neg p_n \equiv \bot \leftarrow p_1 \wedge ... \wedge p_n.$$

  $$: -body_1, ..., body_n,\ (n > 0).$$

Un **programa lógico positivo** es aquel donde hay hechos y reglas ($p \leftarrow q_1, ..., q_n,\ n \geq 0$). En este tipo de problemas P-Completos, podemos el **operador de consecuencias directas** T_P.

Dada una interpretación I y un programa positivo P, $T_P(I)$ nos dará la interpretación directa de ese programa dado la interpretación anterior y se calcula de la siguiente manera:

$$T_P(I) = \{H | (H \leftarrow B) \in P, I \vDash B\} \tag{1}$$

Este operador es monótono y continuo, según el teorema de Knaster & Tarski tiene un punto fijo T_p que se alcanza desde $I = \emptyset$ Es decir, partiendo de $I = \{\emptyset\}$ si hacemos $T_p(I_0), T_p(I_1)..., T_p(I_n)$ hasta que llega un momento en que $T_p(I_{n-1}) = T_p(I_n)$.

Todo programa positivo P tiene uno o varios modelos que están incluidos en todos los demás modelos clásicos llamados **Least Models** $LM(P)$ o **modelos mínimos** que coincide con el punto fijo $T_p(I = \{\emptyset\})$.

Ejemplo:

- Hechos: p. q.
- Reglas: $r \leftarrow p, s.\ s \leftarrow q.\ a \leftarrow b, p.\ b \leftarrow s, a.\ a \leftarrow c.$
- $T_p(I_0 = \{\emptyset\}) = \{p, q\}$, $T_p(I_1) = \{p, q, s\}$, $T_p(I_2) = \{p, q, s, r\} \equiv T_p(I_3) = \{p, q, s, r\}$

  Recordar que $T_p(I)$ es recoger la cabezas que cumplan de las que tengamos el cuerpo, inicialmente, los hechos.

2.4 Separación de la lógica clásica: Negación por defecto

Con la intención de evitar el Frame Problem necesitamos poder saltar a conclusiones en ausencia de información. Para ello, usaremos el **operador** *not* l que significará no hay evidencia para probar el literal l (Aunque eso no impide que en un futuro, si que pueda haberla) .

El **formato de las reglas** pasará a ser $p \leftarrow q_1, ..., q_m$, *not* $q_{m+1}, ...,$ *not* q_n.

Sin embargo, este tipo de negación, la anterior estrategia de coger el modelo más pequeño LM no funcionará: Habrá múltiples modelos mínimos y algunos de ellos no serán apropiados.

Para recoger bien el significado deseado (y no el de las negación clásica) las reglas deben ser direccionales. Por ejemplo, de $\neg v \rightarrow m$ podemos derivar que si tenemos $\neg v$ entonces tenemos m pero no al revés. En el caso de no haber dependencias cíclicas negativas, los programas podrían funcionar por capas pero, no todos los programas tienen reglas independientes unas de otras. Como posible solución tenemos la semántica de modelos estables (ASP).

Semántica de los modelos estables (ASP): Dada una interpretación I se revisarán las reglas negativas quedándose únicamente con aquellas donde las negaciones sean ciertas (y eliminándolas de la regla), lo que simplificará el programa a un programa positivo llamado **reducto** P^I. Si disparamos las reglas $T_{P^I}(I = \emptyset)$ hasta el $LM(P^I)$ y coincide con el supuesto inicial I, este será un modelo estable (Stable Model, SM).

En este tipo de problemas, todos **los modelos estables son clásicos y mínimos**.

Saber si un programa de estas características tiene o no, modelos estables ($SM(P) =_? \emptyset$) es **NP-Completo**.

Ejemplo de programa:

$carne \leftarrow not\ patatas$

$pescado \leftarrow not\ carne$

$patatas \leftarrow pescado$

Modelo Clásico I	Reducto P^I	SM(P^I)
$I_0 = \{carne\}$	carne ← patatas ← *pescado*	$LM(P^{I_0}) = I_0$ stable
$I_1 = \{patatas, pescado\}$	pescado ← patatas ← *pescado*	$LM(P^{I_1}) = I_1$ stable
$I_2 = \{patatas, carne\}$	patatas ← *pescado*	$LM(P^{I_2}) \neq I_2$ no
$I_3 = \{patatas, carne, pescado\}$	patatas ← *pescado*	$LM(P^{I_3}) \neq I_3$ no

Si añadimos la disyunción en la cabeza, lo que nos dejará con reglas con el siguiente formato:

$$H_1, ..., H_n \leftarrow B_1, ..., B_m \tag{2}$$

donde las comas en la cabeza significarán disyunciones.

Para hacer el reducto P^I de los not en la cabeza será al contrario que los del cuerpo: si es falso, desparece y nos quedamos con la regla y, si es verdadero, desaparece la regla. Como hay una disyunción en la cabeza no podemos aplicar el cierre transitivo y debemos elegir los modelos más pequeños. Resolver un programa donde pueda haber disyunciones ($SM(P) =_? \emptyset$) tiene complejidad NP^{NP}. Además, los modelos estables seguirían siendo modelos clásicos pero ya no serían mínimos.

3 Representación relacional y razonamiento

3.1 ASP y lógica relacional

Realmente los programas proposicionales no son lo más interesante de ASP si no, la lógica relacional que se basa en objetos y sus relaciones. Al dominio de esos objetos se le llama **Dominio de Herbrand** y, al contrario que en lógica clásica, si dos elementos son semánticamente distintos también lo serán sintácticamente. Además de este conjunto de hechos, lo que realmente diferencia este tipo de programas de una base de datos normal son las reglas, que nos permiten deducir valores no explícitos (pasando a tener una base de datos deductiva).

- Hechos: spain, germany, cars
- Relaciones: country(spain), bien_consumo(cars)
- Regla: $neighbour(X,Y) :- neighbour(Y,X)$, las variables van en mayúscula.

Estas reglas mediante las **variables** nos permiten referirnos a cualquier objeto del dominio. Desde el punto de vista conceptual, ASP sustituirá esas variables por todos los conjuntos de posibles objetos del dominio (**grounding**), a cada una de las reglas sustituidas (sin variables) se las conoce como **átomos ground**. Esta sustitución tiene lugar antes de hacer solving: buscar los modelos estables. Los modelos estables, en ASP son llamados **conjunto de respuesta**.

- $neighbour(spain, france).\ neighbour(france, germany).$
- $neighbour(X,Y) :- neighbour(Y,X).$
- 2^3 soluciones $= 8$, porque hay 3 átomos

Toda variables que aparezca en una regla debe aparecer al menos una vez en el cuerpo positiva, cuando esto ocurre se dice que es una **variable segura** ya que, ASP puede sustituir valores de información positivos pero no negativos.

Al contrario que los conjuntos extensionales que se definen escribiendo todos los posibles casos, los intensionales de definen usando reglas.

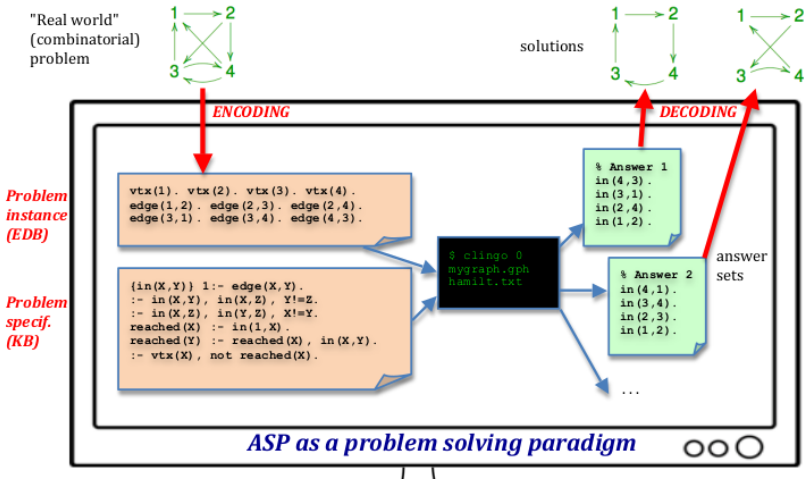
Ejemplo: $2\{friend(X,Y) : person(Y), X! = Y\} :- person(X).$

Fijando persona X, este subconjunto tiene tantos elementos como personas Y hay. Cada valor de X nos da una regla ground diferente mientras que cada valor de Y me esta dando un elemento distinto dentro de un conjunto.

Los programas se suelen partir en:

```
%% Generar soluciones

   in(X,Y) :- edge(X,Y), not out(X,Y). %%Primera opcion
   out(X,Y) :- edge(X,Y), not in(X,Y).

  {in(X,Y): edge(X,Y)}.  %%Segunda opcion

%% Definir
   reached(X) :- in (1,X) %% Empezar a alcanzar desde 1
   reached(Y) :- reached(X), in(X,Y). %% Si ahora X y esta conectado con Y, salta.

%% Testing: Reglas que empiezan por :- y son prohibiciones.
  :- in(X,Y), in(X,Z), Y!=Z. %% De X solo puede llegar a Y
  :- in(X,Z), in(Y,Z), X!=Y. %% De X y Y no se puede llegar a una misma Z
  :- vtx(X), not reached(X) % Prohibido un nodo X no sea alcanzado
```

Desde el punto de vista de como organizamos estos problemas, un punto importante a tener en cuentas es que debemos separar las instancias concretas de cada programa (EDB) de la base de conocimiento (KB), reglas genéricas.



Además de lo anterior, en ASP tenemos más funcionalidades como la **negación fuerte** que, al contrario que la negación por defecto explicada en el apartado 2.4, permite decir si estamos seguros de que un hecho que no es cierto.

```
%% Si no hay evidencia que no haya tren cruzo
cross :- no_train.
:- train, no_train.
%% Cuando este 100% seguro de que no hay tren cruzo
cross:- -train. %% Cruzo solo si tengo la certeza de que no hay tren
:- train, -train %% Prohibido que haya y no haya tren
```

Otra funcionalidad sería el pooling que nos permite abreviar hechos en un mismo átomo. También disponemos de constantes, constructores y funciones.

```
%% CONSTANTES
#const numhouses=5.
house(1..numhouses).

%% CONSTRUCTORES
owner(person(bill,gates), microsoft).
owner(company(inditex), zara).
owner(person(jeff,bezos), amazon).
family(Y) :- owner( person(_,Y), _). %% Nos da todos los apellidos
```

En el caso de las funciones hay que tener en cuenta que se actúa sobre conjuntos.

```
income(may,10).
income(jun,10).
income(jan,5).
income(feb,3).
total(S) :- #sum{X: income(M,X)} = S. %Suma todos los X tal que sean income(M,X)
%% S seria 18 (no 28) ya que el conjuntos no pueden tener repetidos {3,5,10}
```

3.2 ASP Temporales

Los problemas más típicos a resolver como se comenta en el apartado 1 son dinámicos y se pueden resolver mediante ASP teniendo en cuenta transiciones entre estados (mediante una variable extra). Estos problemas están formados por: fluentes, acciones, estados (configuración de los valores de los fluentes) y situación (momento en el tiempo).

Con clingo podemos simular temporalidad, usando una variable extra que simule temporalidad pero, en vez de eso, podemos usar telingo. Es igual a clingo pero el programa se divide en secciones:

```
# program initial. % Estado inicial (t = 0)
# program dynamic. % Reglas que ocurren entre una transicion
# program always. % Se tiene que cumplir en todas las situaciones (t=0...n-1)
# program final. % En el ultimo paso t = n-1
```

Además para referirnos a determinados estados usaremos:

- ' para referirnos al estado anterior al actual (t-1)
- _ para referirnos al estado inicial (t=0)

Existe una plantilla para facilitar mantener un fluente, las acciones o la inercia:

```
#program dynamic.
%% o(A): La accion A ha ocurrido (occurs)
%% h(F,V): El fluente F tiene el valor V (holds)
%% c(F,V): El fluente F ha sido causado para tomar el valor V (caused)
%% Inercia:
h(F,V) :- 'h(F,V), not c(F). %Si no hay cambio, todo se mantiene
h(F,V) :- c(F,V). %Si si ha cambiado respeta nuevo valor
c(F) :- c(F,V).
```

3.3 Numero de soluciones y reglas ground

3.3.1 Numero de soluciones

```
#const n=3.
digito(1..n). fila(1..n). columna(1..n).
#show celda/3.
1 {celda(X,Y,D): digito(D)} 1 :- fila(X),columna(Y).
```

¿Qué tipo de soluciones generaría este programa y cuántas generaría? Este programa asigna un número entre 1 y 3 a cada celda. Esto permite repetir números en la misma fila o columna. Como hay $33 = 9$ celdas y, para cada una, podemos elegir entre 3 posibilidades, el número de soluciones sería 3^9.

```
color(rojo;verde;azul).      pais(fr;de;be).vecino(fr,de).              vecino(fr,be).  vecino(be,de).
vecino(X,Y) :- vecino(Y,X).
#show pinta/2.
1 { pinta(X,C) : color(C) } 1 :- pais(X).
:- vecino(X,Y), pinta(X,C), pinta(Y,C)
```

¿Qué tipo de soluciones generaría este programa y cuántas generaría? Como tenemos justo 3 países y todos son vecinos de todos. N. soluciones = permutaciones de 3 colores, es decir, $3! = 6$.

3.3.2 Reglas Ground

Cómo calcular cuantas reglas ground se generan en los programas:

- Reglas que generan $relaciona(X,Y) : relacionB(Y) : -relacionC(X)$. Cada valor de X nos da una regla ground diferente mientras que cada valor de Y da un elemento distinto dentro de un conjunto.

  Ej1: En este caso de la regla de la línea 6 se generarían 3 reglas ground, una por pais.

```
1         color(rojo;verde;azul).
2         pais(fr;de;be).
3         vecino(fr,de). vecino(fr,be). vecino(be,de).
4         vecino(X,Y) :- vecino(Y,X).
5         #show pinta/2.
6         1 { pinta(X,C) : color(C) } 1 :- pais(X).
7         :- vecino(X,Y), pinta(X,C), pinta(Y,C)
```

  Ej2: Se generaría por la regla 6, 3x3 = 9 reglas ground, una por celda.

```
1             #const n=3.
2             digito(1..n).
3             fila(1..n).
4             columna(1..n).
5             #show celda/3.
6             1 {celda(X,Y,D): digito(D)} 1 :- fila(X),columna(Y).
```

- Prohibiciones:

  EJ1 - La prohibición generaría: (3!)x3, (3!=6 permutaciones) 6 hechos vecino(X,Y) por los tres colores.

```
1         color(rojo;verde;azul).
2         pais(fr;de;be).
3         vecino(fr,de). vecino(fr,be). vecino(be,de).
4         vecino(X,Y) :- vecino(Y,X).
5         #show pinta/2.
6         1 { pinta(X,C) : color(C) } 1 :- pais(X).
7         :- vecino(X,Y), pinta(X,C), pinta(Y,C)
```

Ej2- La primera restricción, por cada fila X y por cada dígito D toma pares de números Y¡Y1 entre 1 y 3. Esto sólo genera tres casos:Y=1, Y1=2; Y=1, Y1=3; y Y=2,Y1=3. En total, 3 filas 3 dígitos 3 pares = 27 reglas ground. La segunda restricción es totalmente simétrica y genera otros 27 casos.

```
1         #const n=3.
2         digito(1..n). fila(1..n). columna(1..n).
3         #show celda/3.
4         1 {celda(X,Y,D): digito(D)} 1 :- fila(X),columna(Y)
5         :- celda(X,Y,D), celda(X,Y1,D), Y<Y1.
6         :- celda(X,Y,D), celda(X1,Y,D), X<X1.
```

- Reglas:

  Ej1 - El programa tiene 3 hechos para fila, 3 hechos para columna, y 3 hechos para digito del ejemplo anterior.

  Ej2 - Las primeras dos reglas son en total 6 y hay que tenerlas en cuenta. Y la regla vecino genera 3!=6 porque son las posibles permutaciones.

```
1         color(rojo;verde;azul).
2         pais(fr;de;be).
3         vecino(fr,de). vecino(fr,be). vecino(be,de).
4         vecino(X,Y) :- vecino(Y,X).
5         #show pinta/2.
6         1 { pinta(X,C) : color(C) } 1 :- pais(X).
7         :- vecino(X,Y), pinta(X,C), pinta(Y,C)
```

  Ej3 - ¿Cuantos hechos se generan para fichas(F)?

```
1         #show asigna/2.
2         digito(0..6).
3         ficha( par(A,B) ) :- digito(A), digito(B), A<=B.jugador(1..4).
```

  Si tenemos n dígitos diferentes, el número de fichas viene dado por combinaciones con repetición de esos n dígitos tomados de dos en dos $\binom{n+2-1}{n} = \frac{n(n+1))}{2}$.