Intelligenza Artificiale – Lezione 4
17 aprile 2023

Programmazione Logica e PROLOG

Rispetto ai linguaggi di programmazione classici come C o Pascal, la logica rende possibile esprimere relazioni in modo elegante, compatto e dichiarativo. I dimostratori automatici di teoremi sono persino in grado di decidere se una base di conoscenza implica logicamente una query (tesi). Il calcolo delle prove e la conoscenza immagazzinata nella knowledge base sono rigorosamente separati. Una formula descritta nella forma normale di clausola può essere utilizzata come dato di input per qualsiasi dimostratore di teoremi, indipendentemente dal calcolo dimostrativo utilizzato. Questo è di grande valore per il ragionamento e la rappresentazione della conoscenza.
Se si desidera implementare algoritmi, che inevitabilmente hanno componenti procedurali, una descrizione puramente dichiarativa è spesso insufficiente. Robert Kowalski, uno dei pionieri della programmazione logica, ha sottolineato questo punto con la formula

Algoritmo = Logica + Controllo.

Questa idea è stata realizzata nel linguaggio PROLOG. PROLOG è utilizzato in molti progetti, principalmente nell'IA e nella linguistica computazionale. Daremo ora una breve introduzione a questo linguaggio e ne presenteremo i concetti più importanti.

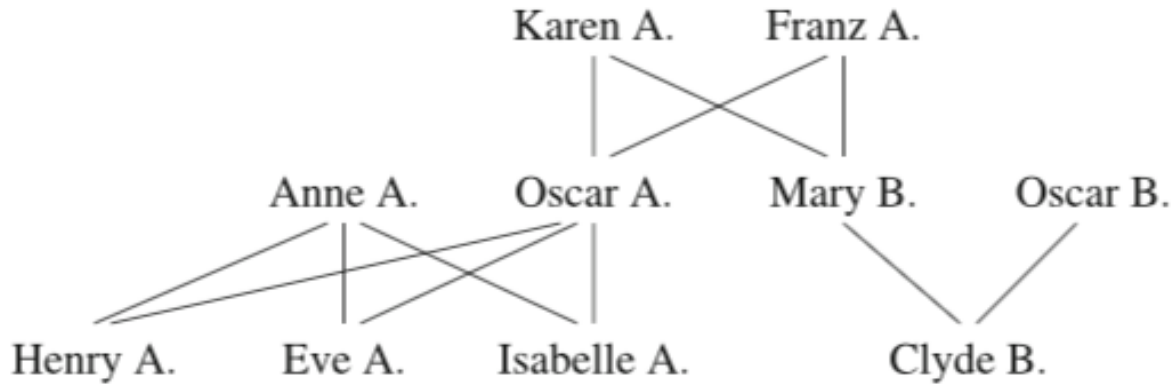
La sintassi del linguaggio PROLOG consente solo clausole di Horn. La notazione logica e la sintassi di PROLOG sono giustapposte nella tabella qui sotto:

PL1 / clause normal form	PROLOG	Description
$(\neg A_1 \vee \ldots \vee \neg A_m \vee B)$	`B :- A1, ... , Am.`	Rule
$(A_1 \wedge \ldots \wedge A_m) \Rightarrow B$	`B :- A1, ... , Am.`	Rule
A	`A.`	Fact
$(\neg A_1 \vee \ldots \vee \neg A_m)$	`?- A1, ... , Am.`	Query
$\neg(A_1 \wedge \ldots \wedge A_m)$	`?- A1, ... , Am.`	Query

A_1,..., A_m, A, B sono letterali. I letterali sono, come in PL1 (logica del prim'ordine), costituiti da simboli di predicato con termini come argomenti. Come possiamo vedere nella tabella sopra, in PROLOG non ci sono negazioni in senso stretto logico perché il segno (positivo o negativo) di un letterale è determinato dalla sua posizione nella clausola.

La maggior parte dei moderni sistemi PROLOG funziona con un interprete basato sulla Warren Abstract Machine (WAM). Nel 1983, David H. D. Warren progettò una macchina astratta per l'esecuzione di PROLOG composta da un'architettura di memoria e un set di istruzioni. Questo progetto divenne noto come Warren Abstract Machine (WAM) ed è diventato di fatto lo standard per i compilatori Prolog. Il codice sorgente PROLOG viene compilato nel cosiddetto codice WAM, che viene quindi interpretato dalla WAM. Le implementazioni più veloci di un WAM gestiscono fino a 10 milioni di inferenze logiche al secondo (LIPS: logic inferences per second) su un PC con processore da 1 GHz.

Riprendiamo l'esempio dell'albero genealogico, qui sotto mostrato in figura e poi codificato sotto forma di formule della logica del prim'ordine.



$$
\begin{aligned}
KB \equiv\ & female(karen) \wedge female(anne) \wedge female(mary) \\
& \wedge female(eve) \wedge female(isabelle) \\
& \wedge child(oscar, karen, franz) \wedge child(mary, karen, franz) \\
& \wedge child(eve, anne, oscar) \wedge child(henry, anne, oscar) \\
& \wedge child(isabelle, anne, oscar) \wedge child(clyde, mary, oscarb) \\
& \wedge (\forall x\, \forall y\, \forall z\, child(x, y, z) \Rightarrow child(x, z, y)) \\
& \wedge (\forall x\, \forall y\, descendant(x, y) \Leftrightarrow \exists z\, child(x, y, z) \\
& \vee (\exists u\, \exists v\, child(x, u, v) \wedge descendant(u, y))).
\end{aligned}
$$

Trascurando i predicati sul genere degli individui, ecco la KB sotto forma di programma PROLOG:

```
 1 child(oscar,karen,frank).
 2 child(mary,karen,frank).
 3 child(eve,anne,oscar).
 4 child(henry,anne,oscar).
 5 child(isolde,anne,oscar).
 6 child(clyde,mary,oscarb).
 7
 8 child(X,Z,Y) :- child(X,Y,Z).
 9
10 descendant(X,Y) :- child(X,Y,Z).
11 descendant(X,Y) :- child(X,U,V), descendant(U,Y).
```

La conversione da formule della logica del prim'ordine a istruzioni PROLOG è un procedimento complesso ma molto simile a quanto abbiamo fatto noi per ricavare le clausole per fare la risoluzione. Le uniche condizioni aggiuntive sono che possiamo ammettere nella KB riformulata per PROLOG solo quelle clausole che sono di Horn (zero o al massimo un letterale positivo, tutti

gli altri negati). In particolare, dalle clausole che derivano dalla formula sul predicato "Descendant" possiamo prendere solo quelle che derivano dalla sottoformula con $\Leftarrow$, in cui *Descendant(x,y)* diventa un letterale positivo e tutti gli altri letterali *Child(x,y,z)*, *Child(x,u,v)*, *Descendant(u,y)* sono con negazione.

Caricate le istruzioni PROLOG in un interprete PROLOG (es.: https://swish.swi-prolog.org/, cliccare su "Program").
Poi nella finestra delle query, inserite la query:

?- child(eve,oscar,anne).

L'interprete vi darà come risposta "true", perché la query è effettivamente una conseguenza logica della KB e l'interprete riesce a lavorare con la clausole di Horn per dimostrarne la verità.

Inoltre, l'interprete da un avvertimento di "singleton variable" per Z e V, per aiutare il programmatore con due errori comuni: errori di ortografia nelle variabili e dimenticarsi di usare o associare una variabile. L'interprete indica che ci sono una o più variabili che appaiono solo una volta all'interno di una clausola. La prima istanza di una variabile dà sempre luogo a un binding riuscito. Se questa associazione non viene utilizzata da nessuna parte, non accade nulla. Semplicemente l'interprete vuole accertarsi che queste variabili non utilizzate non siano un errore di ortografia oppure sono poi usate in altre clausole che l'utente si è dimenticato di inserire.

Tornando al risultato della query, come nasce questa risposta "true"? Per la query

?- child (eve, oscar, anne)

ci sono sei fatti e una regola con lo stesso predicato nella testa della clausola. Ora viene tentata l'unificazione tra la query e ciascuno dei letterali *complementari* nei dati di input in ordine di occorrenza. Se una delle alternative fallisce, ciò si traduce in un backtracking all'ultimo punto di diramazione e viene testata l'alternativa successiva. Poiché l'unificazione fallisce con ogni fatto (i nomi delle persone sono costanti e non possono essere sostituiti, quindi se non troviamo il letterale identico a quello della query non c'è unificazione), la query viene unificata con la regola ricorsiva nella riga 8. Ora il sistema tenta di risolvere l'obiettivo secondario child(eve, anne, oscar), che riesce con la terza alternativa.

La query
? - descendant(clyde, karen)
invece, non ottiene risposta. La ragione di ciò è la clausola nella riga 8, che specifica la simmetria del predicato figlio. Questa clausola si chiama in modo ricorsivo senza possibilità di risoluzione. Questo problema può essere risolto con il seguente nuovo programma (i fatti sono stati omessi in figura ma nel programma vanno aggiunti tutti i fatti, ossia le clausole singole child(_,_,_)).

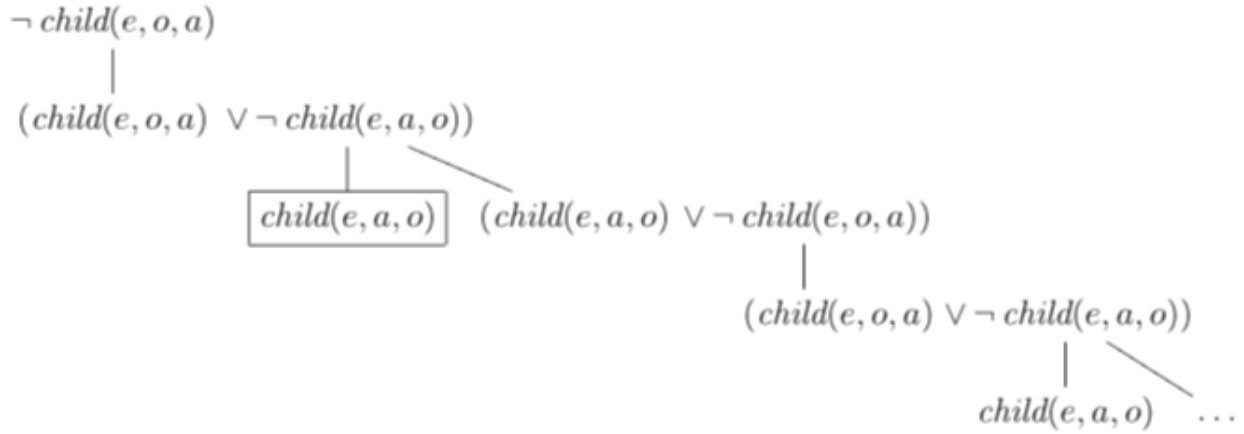
```
descendant(X,Y) :- child(X,Y,Z).
descendant(X,Y) :- child(X,Z,Y).
descendant(X,Y) :- child(X,U,V), descendant(U,Y).
```

Ora, però, la query
?- child(eve,oscar,anne)
riceve risposta "false" perché la simmetria del predicato child nelle ultime due variabili non è più data.

Una soluzione a entrambi i problemi si trova nel programma

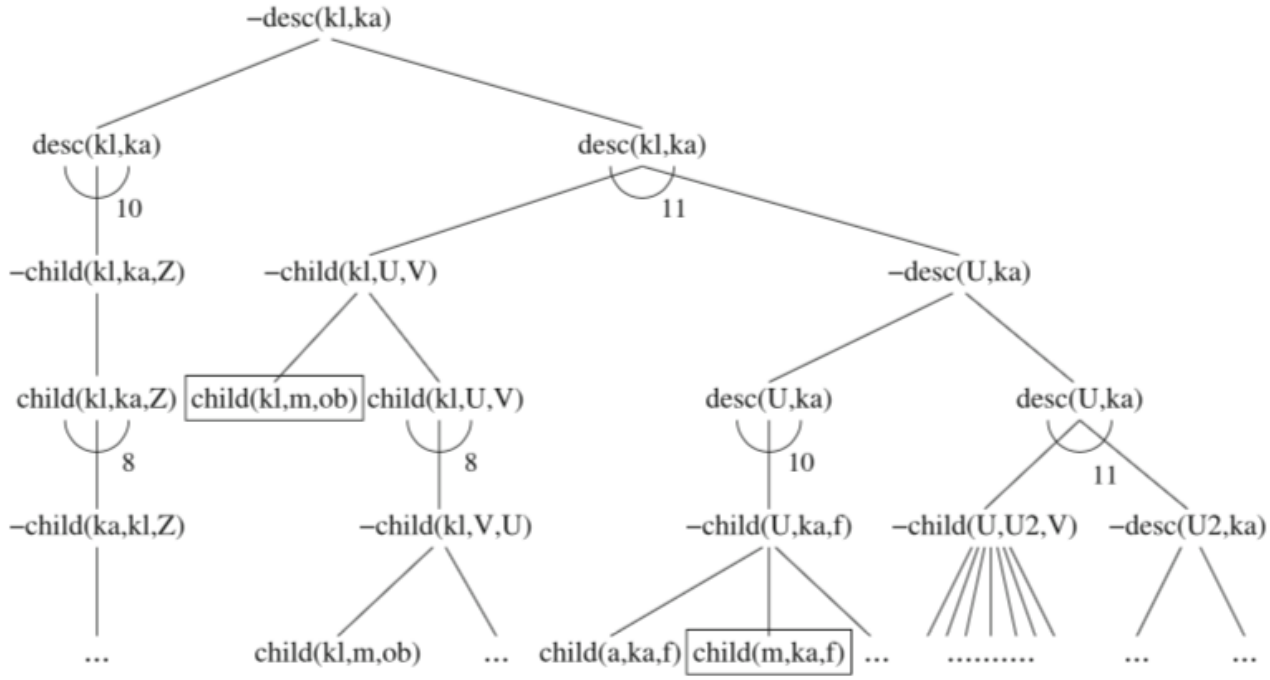
```
 1 child_fact(oscar,karen,franz).
 2 child_fact(mary,karen,franz).
 3 child_fact(eva,anne,oscar).
 4 child_fact(henry,anne,oscar).
 5 child_fact(isolde,anne,oscar).
 6 child_fact(clyde,mary,oscarb).
 7
 8 child(X,Z,Y) :- child_fact(X,Y,Z).
 9 child(X,Z,Y) :- child_fact(X,Z,Y).
10
11 descendant(X,Y) :- child(X,Y,Z).
12 descendant(X,Y) :- child(X,U,V), descendant(U,Y).
```

Introducendo il nuovo predicato child_fact per i fatti, il predicato child non è più ricorsivo.
Tuttavia, il programma non è più elegante e semplice come la prima variante, logicamente corretta, presentata in precedenza, che conduce al loop infinito.
Il programmatore PROLOG deve, proprio come in altri linguaggi, prestare attenzione all'elaborazione ed evitare loop infiniti.
PROLOG è solo un linguaggio di programmazione e non un dimostratore di teoremi.
Dobbiamo distinguere qui tra semantica *dichiarativa* e *procedurale* dei programmi PROLOG.
La semantica dichiarativa è data dall'interpretazione logica delle clausole di Horn.
La semantica procedurale, al contrario, è definita dall'esecuzione del programma PROLOG, che ora desideriamo osservare più in dettaglio.
L'esecuzione della prima variante del programma con la query child(eve, oscar, anne) è rappresentata nella figura sotto come un albero di ricerca. L'esecuzione inizia in alto a sinistra con la query. Ciascun segmento rappresenta un possibile passo di risoluzione SLD ("Selection-rule-driven Linear resolution for Definite clauses", risoluzione lineare basata su regole di selezione per clausole definite) con un letterale complementare unificabile. Mentre l'albero di ricerca diventa infinitamente profondo dalla regola ricorsiva, l'esecuzione di PROLOG termina perché i fatti si verificano prima della regola nei dati di input.



In figura: l'albero di ricerca PROLOG per child(eve,oscar,anne).

Con la query descendant(clyde, karen), invece, l'esecuzione di PROLOG non termina. Possiamo vederlo chiaramente nell'albero and-or presentato in figura qui sotto. In questa rappresentazione i rami, rappresentati da segmenti che convergono in un punto e intersecati da un arco, conducono dalla testa di una clausola agli obiettivi secondari. Poiché tutti gli obiettivi secondari di una clausola devono essere risolti, questi sono rami "and". Tutti gli altri rami sono rami "or", di cui almeno uno deve essere unificabile con i suoi nodi padre. I due fatti racchiusi in rettangoli rappresentano la soluzione alla domanda. L'interprete PROLOG non termina qui, però, perché lavora utilizzando una ricerca in profondità ("depth-first") con backtracking e quindi sceglie prima il percorso infinitamente profondo all'estrema sinistra.



Elementi procedurali e controllo dell'esecuzione

Come abbiamo visto nell'esempio della relazione familiare, è importante controllare l'esecuzione di PROLOG. Evitare in particolare il backtracking non necessario può portare a grandi aumenti di efficienza. A tal fine esiste lo strumento del "taglio". Inserendo un punto esclamativo in una clausola, possiamo impedire il backtracking in quel punto. Nel seguente programma, il predicato max (X, Y, Max) calcola il massimo dei due numeri X e Y.

```
1 max(X,Y,X) :- X >= Y.
2 max(X,Y,Y) :- X < Y.
```

Se si applica il primo caso (prima clausola), il secondo non verrà raggiunto.
D'altra parte, se il primo caso non si applica, la condizione del secondo caso è (necessariamente) vera (perché i due casi sono mutuamente esclusivi ed esaustivi), il che significa che non è necessario verificarlo. Ad esempio, nella query
? - max (3,2,Z), Z > 10.
viene utilizzato il backtracking perché Z = 3, e la seconda clausola è testata per max. Tale procedura però è destinata a fallire. Quindi tornare indietro su questo punto non è necessario.

Possiamo ottimizzare il tutto con un taglio:

```
1 max(X,Y,X) :- X >= Y, !.
2 max(X,Y,Y).
```

In questa variante, la seconda clausola viene chiamata solo se è veramente necessario, cioè se la prima clausola fallisce. Tuttavia, questa ottimizzazione rende il programma più difficile da capire.
Potete seguire la differenza nell'esecuzione dell'interprete PROLOG attivando la modalità debug: scrivere la query nella finestra delle query, poi cliccare su "Solutions" e poi su "Debug (trace)"; una volta lanciata la computazione cliccando su "Run", cliccare sull'icona "step into" per seguire passo a passo la computazione.
Un'altra possibilità per il controllo dell'esecuzione è il predicato "fail" pronto all'uso, che non è mai vero. Nell'esempio della relazione familiare possiamo semplicemente stampare tutti i bambini e i loro genitori con la query:
?- child_fact(X,Y,Z), write(X), write('is a child of '), write(Y), write(' and '), write(Z), write('.'), nl, fail.
Che cosa succede se non inseriamo "fail" alla fine di questa query?

Liste

Come linguaggio di alto livello, PROLOG ha, come il linguaggio LISP, il comodo tipo di dati "lista generica".
Una lista con gli elementi a, 2, 2, b, 3, 4, 5 ha la forma
[a, 2,2, b, 3,4,5]
Il costrutto [H | T] separa il primo elemento (H, head, testa) dal resto (T, tail, coda) della lista.
Con la base di conoscenza:
list([a, 2,2, b, 3,4,5]).
PROLOG visualizza a seguito della query ? - list ([H | T]) quanto segue:
H = A
T = [2, 2, b, 3, 4, 5]

Un esempio elegante di elaborazione delle liste è la definizione del predicato
append (X, Y, Z)
per aggiungere la lista Y alla lista X.
Il risultato viene salvato in Z.
Il programma PROLOG è come segue:

```
1 append([],L,L).
2 append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

Questa è una descrizione logica (ricorsiva) del fatto che L3 risulta dall'aggiunta di L2 a L1. Allo stesso tempo, tuttavia, questo programma compie anche un lavoro quando viene chiamato.
La chiamata

? - append ([a, b, c], [d, 1,2], Z).

restituisce la sostituzione Z = [a, b, c, d, 1, 2],

proprio come la chiamata
? - append (X, [1,2,3], [4,5,6,1,2,3]).

restituisce la sostituzione X = [4, 5, 6].

L'inversione dell'ordine degli elementi di una lista può anche essere descritta elegantemente e simultaneamente programmata dal predicato ricorsivo:

```
1 nrev([],[]).
2 nrev([H|T],R) :- nrev(T,RT), append(RT,[H],R).
```

che riduce l'inversione di una lista all'inversione di una lista che è più piccola di un elemento.
In effetti, questo predicato è molto inefficiente a causa della chiamata append.
Questo programma è noto come "naïve reverse" e viene spesso utilizzato come benchmark degli ambienti PROLOG.
Le cose vanno meglio quando si procede utilizzando un contenitore temporaneo, noto come accumulatore, come segue:

List	Accumulator
[a,b,c,d]	[]
[b,c,d]	[a]
[c,d]	[b,a]
[d]	[c,b,a]
[]	[d,c,b,a]

Il programma PROLOG corrispondente è come segue:

```
1 accrev([],A,A).
2 accrev([H|T],A,R) :- accrev(T,[H|A],R).
```

Per ottenere l'inversa di una lista scrivere la query:
accrev([<lista da invertire>],[],[<lista in cui il risultato viene costruito>]).