Intelligenza artificiale e laboratorio

Appunti delle lezioni di
Paolo Alfano

Note Legali

Liberatoria, aggiornamenti, segnalazione errori:

Quest'opera viene pubblicata in formato elettronico senza alcuna garanzia di correttezza del suo contenuto. Il documento, nella sua interezza, è opera di Paolo Didier Alfano
e viene mantenuto e aggiornato dallo stesso, a cui possono essere inviate eventuali segnalazioni all'indirizzo *paul15193@hotmail.it*
Ultimo aggiornamento: 05 settembre 2017

Indice

20/02/2017

1 Introduzione al Prolog

1.1 Le basi

Il linguaggio che andremo ad utilizzare e' il Prolog.
Il Prolog e' un linguaggio dichiarativo che descrive la situazione usando dei *fatti* e delle *regole*. Tramite regole e fatti possiamo verificare la raggiungibilita', o validita' di un *goal*.

Esempio 1 : supponiamo di definire la funzione gatto(x) che indica che x e' un gatto e miagola(y) ovvero che y miagola. Supponiamo inoltre di essere a conoscenza del seguente fatto:
gatto(Tom)
Supponiamo inoltre che valga la seguente regola:
$\forall x : gatto(x) \Rightarrow miagola(x)$
Possiamo concludere che miagola(Tom)? Possiamo dirlo se il goal e' raggiungibile.

La raggiungibilita' di un certo goal e' verificata utilizzando le regole della logica classica. Vediamone un altro esempio

Esempio 2 : consideriamo la funzione piuCalorico(x,y) tramite cui indichiamo che x e' un cibo maggiormente calorico di y. Supponiamo inoltre di essere a conoscenza dei seguenti fatti

$piuCalorico(pancetta, wurstel).$ $piuCalorico(wurstel, banana).$
$piuCalorico(banana, verza).$ $piuCalorico(banana, cetriolo).$

Notiamo che per la prima volta abbiamo usato la sintassi del Prolog in cui segnaliamo la fine di un fatto tramite un punto. Approfittiamone per introdurre anche la sintassi del goal, ovvero la sintassi dell'interrogazione.
$? - piuCalorico(verza, wurstel).$
$false$
Quello che ci stiamo chiedendo e' se dall'insieme dei fatti possiamo ricavare il goal. In questo caso la risposta che otteniamo sara' negativa, segnalata tramite il valore *false*.

Possiamo anche utilizzare le variabili in Prolog come segue:

Esempio 3 : volendo sapere quali cibi siano piu' calorici del wurstel, effettuiamo l'interrogazione seguente.
$? - piuCalorico(Quale, wurstel).$
$Quale = pancetta$

A questo livello pero' Prolog non e' capace di inferire nuovi fatti. Ad esempio la transitivita' non e' implementata e questo porta delle contraddizioni con il senso comune. Vediamolo con il breve esempio che segue

Esempio 4 : supponendo di possedere ancora i fatti dell'esempio 2, potremmo pensare di chiedere
$? - piuCalorico(pancetta, verza)$
Intuitivamente ci aspetteremmo che la risposta sia $true$ visto che in termini di calorie $pancetta > wurstel > banana > verza$. Eppure la risposta di Prolog e' $false$.

Come dicevamo questa risposta e' giustificata dal fatto che Prolog non controlla la transitivita'. Per farlo abbiamo due metodi

1. Aggiungere manualmente le condizioni implicate, ovvero
   $piuCalorico(pancetta, banana)$, $piuCalorico(pancetta, verza) \ldots$
   ma chiaramente questa soluzione e' pessima
2. Aggiungere una regola di deduzione del Prolog. Vediamolo tramite il prossimo esempio

Esempio 5 : supponiamo di scrivere una nuova regola $ePiuCal(X, Y)$ in Prolog come segue

```
//informazione esplicita
ePiuCal(X, Y) :- piuCalorico(X, Y).
//proprieta' transitiva
ePiuCal(X, Y) :-
   piuCalorico(X, Z), ePiuCal(Z, Y)
```

La sintassi :- indica che $ePiuCal(X, Y)$ e' verificato se e' vero che $piuCalorico(X, Y)$, in pratica indica la proprieta' logica:

$$piuCalorico(X, Y) \Rightarrow ePiuCal(X, Y)$$

Quello che stiamo dicendo e' che X e' piu calorico di Y se abbiamo l'informazione esplicita a riguardo oppure se vale la transitivita'.
Notiamo quindi, che la programmazione in Prolog e' di tipo ricorsivo, infatti stabiliamo dei casi base e un'insieme di regole ricorsive. Sfruttando i casi base e le regole ricorsive cerchiamo di effettuare pattern-matching.
A questo punto, andando a chiedere
$? - ePiuCal(pancetta, verza)$.
$true$

Come ragiona l'interprete Prolog per verificare la raggiungibilita' del goal? In Prolog diciamo che la ricerca e' guidata dal goal, infatti partiamo proprio dal goal:
$ePiuCal(pancetta, verza)$
Supponiamo che il codice nel suo complesso sia

```
piuCalorico(pancetta, wurstel).
piuCalorico(wurstel, banana).
piuCalorico(banana, verza).
piuCalorico(banana, cetriolo).

ePiuCal(X, Y) :- piuCalorico(X, Y).
ePiuCal(X, Y) :-
   piuCalorico(X, Z), ePiuCal(Z, Y)
```

Per verificare il goal andiamo a scandire il codice. Ignoriamo i fatti che nulla hanno a che vedere con $ePiuCal(X, Y)$. Quando raggiungiamo la prima definizione di ePiuCal vediamo se vale che $piuCalorico(X, Y)$ ovvero se vale che $piuCalorico(pancetta, verza)$ ma come gia' sappiamo tale proprieta' non vale poiche' non usa la transitivita'.
Proseguiamo oltre e troviamo una nuova definizione di $ePiuCal(X, Y)$. Tramite essa possiamo dire che il goal e' verificato se sono verificate le due proprieta' che compongono la regola, ovvero $piuCalorico(X, Z), ePiuCal(Z, Y)$. Dunque andando a sostituire le variabili possiamo dire che il goal e' verificato se e' verificato il sottogoal:
$piuCalorico(pancetta, Z), ePiuCal(Z, verza)$
A questo punto Prolog cerca di verificare questo nuovo sottogoal. A tale scopo setta:
$X = pancetta$, $Y = verza$ e $Z = wurstel$
Allora abbiamo che $piuCalorico(pancetta, wurstel)$ e' verificata poiche' tale informazione e' contenuta esplicitamente nell'insieme dei nostri fatti. Resta da verificare $ePiuCal(Z, verza)$ ovvero $ePiuCal(wurstel, verza)$.
Anche in questo caso la prima definizione di $ePiuCal$ non ci aiuta, pero' la seconda definizione ci soccorre nuovamente. Infatti settando le variabili:
$X = wurstel$, $Y = verza$ e $Z = banana$
abbiamo che $ePiuCal(wurstel, verza)$ e' verificata se
$piuCalorico(wurstel, Z), ePiuCal(Z, verza)$.
Anche in questo caso $piuCalorico(wurstel, banana)$ e' direttamente verificato dai fatti. Dobbiamo allora verificare $ePiuCal(banana, verza)$
Stavolta per fortuna la prima definizione di $ePiuCal$ ci viene in aiuto infatti poiche' $piuCalorico(banana, verza)$ e' direttamente presente tra i fatti, allora la definizione $ePiuCal(X, Y)$ e' verificata.
La dimostrazione e' conclusa e abbiamo mostrato l'intero procedimento che l'interprete segue per andare a verificare che la pancetta e' piu' calorica della verza. L'intero procedimento e' riassumibile tramite un albero che rappresentiamo in Figura 1

22/02/2017

A questo punto possiamo illustrare una nuova funzionalita' del linguaggio Prolog.

Esempio 6 : avendo a disposizione la stessa base di conoscenza degli esempi precedenti potremmo pensare di chiederci quali siano i cibi piu' calorici

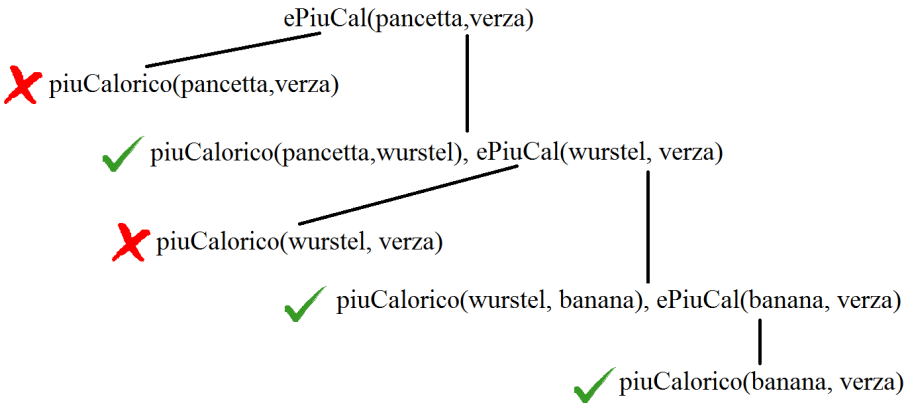

Figura 1: L'albero che rappresenta i passi eseguiti dall'interprete Prolog per verificare $ePiuCal(pancetta, verza)$

del cetriolo. L'interrogazione in questo caso e':

$? - ePiuCal(X, cetriolo)$
$X = banana$
Notiamo che viene restituito un solo risultato, mentre ci aspetteremmo che anche wurstel e pancetta vengano restituiti. Per elencarli e' sufficiente digitare il simbolo ";".

Esempio 7 : possiamo anche fare richieste piu' complesse del tipo:

$? - ePiuCal(pancetta, X), ePiuCal(X, verza)$
$X = wurstel$
In questo caso chiediamo l'insieme degli alimenti meno calorici della pancetta e piu' calorici della verza(anche in questo caso e' stato restituito un solo risultato anche se abbiamo piu' di un risultato valido)

Un'altra funzionalita' interessante del Prolog e' *trace*. Inserendo da linea di comando tale keyword(come sempre seguita da un punto), eseguendo una query ne otteniamo la traccia di esecuzione seguita dal risultato.

Esempio 8 : eseguendo la query che gia' avevamo analizzato $ePiuCal(pancetta, verza)$, otteniamo il risultato seguente:

```
[trace] ?- ePiuCal(pancetta, verza).
   Call: (8) ePiuCal(pancetta, verza) ? creep
   Call: (9) piuCalorico(pancetta, verza) ? creep
   Fail: (9) piuCalorico(pancetta, verza) ? creep
   Redo: (8) ePiuCal(pancetta, verza) ? creep
   Call: (9) piuCalorico(pancetta, _770) ? creep
   Exit: (9) piuCalorico(pancetta, wurstel) ? creep
```

```
   Call: (9) ePiuCal(wurstel, verza) ? creep
   Call: (10) piuCalorico(wurstel, verza) ? creep
   Fail: (10) piuCalorico(wurstel, verza) ? creep
   Redo: (9) ePiuCal(wurstel, verza) ? creep
   Call: (10) piuCalorico(wurstel, _770) ? creep
   Exit: (10) piuCalorico(wurstel, banana) ? creep
   Call: (10) ePiuCal(banana, verza) ? creep
   Call: (11) piuCalorico(banana, verza) ? creep
   Exit: (11) piuCalorico(banana, verza) ? creep
   Exit: (10) ePiuCal(banana, verza) ? creep
   Exit: (9) ePiuCal(wurstel, verza) ? creep
   Exit: (8) ePiuCal(pancetta, verza) ? creep
true .
```

Notiamo che il percorso seguito da trace e' fondamentalmente quello che abbiamo mostrato in Figura 1. Infatti grazie alla seconda definizione di ePiuCal viene introdotta ogni volta una variabile(salvata con il nome _770) che viene poi istanziata ad uno dei valori possibili.
Studiamo brevemente il significato di alcune keyword:

- Call: indica l'esecuzione di una query
- Fail: indica che non e' stata trovata un'associazione per quella query, dunque c'e' stato un fallimento
- Exit: indica che e' stato trovato un match per quella query, dunque c'e' stato un successo su di essa
- Redo: indica che e' stata trovata un'altra definizione di una query fallita precedentemente

Il valore indicato tra parentesi indica il livello di profondita' della query.

Notiamo che da tutto cio' che abbiamo detto sembra emergere l'idea che sia importante l'ordine con cui vengono scritte le regole. Questo perche' l'interprete Prolog scandisce il documento dall'inizio verso la fine cercando dei fatti o delle regole applicabili. Appena trova una regola applicabile la espande.
Regole introdotte in modo non corretto potrebbero dare luogo a spiacevoli conseguenze. Nel seguito vedremo un esempio che genera un loop per comprendere meglio quanto appena detto.

1.2 Vincoli

Per introdurre i vincoli sulle variabili, mostriamo un esempio

Esempio 1 : supponiamo di avere a disposizione un albero genealogico che viene descritto tramite una serie di fatti del tipo:

$$genitore(X, Y)$$

Con cui indichiamo che X e' genitore di Y.
Vogliamo allora creare una regola che ci permetta di individuare i fratelli all'interno dell'albero. Sapendo che due persone sono fratelli se hanno lo stesso padre e la stessa madre, allora potremmo scrivere come segue la nostra regola:

```
fratello(X, Y):-
   genitore(Padre, X),
   genitore(Padre, Y),
   genitore(Madre, X),
   genitore(Madre, Y).
```

Sebbene questa regola possa sembrare intuitivamente giusta possiede due problemi

1. Possiamo essere fratelli di noi stessi poiche' nessuno ha detto al linguaggio che X e Y debbano essere distinti
2. Fratelli e fratelli unilaterali[1] sono considerati uguali poiche' nel momento in cui esista un genitore comune(caso dei fratelli unilaterali) Prolog istanzia sia Padre che Madre a tale genitore comune. Anche in questo caso nessuno ha specificato al linguaggio che Padre e Madre devono essere distinti

Come possiamo allora introdurre i vincoli? La sintassi per indicare che X e' diverso da Y e' $X \backslash == Y$. A questo punto possiamo scrivere una regola per individuare i fratelli e un'altra per trovare i fratelli unilaterali.

Esempio 2 : la regola corretta per i fratelli e per i fratelli unilaterali e':

```
fratello(X,Y):-
   genitore(Padre,X),
   genitore(Padre,Y),
   X \== Y,
   genitore(Madre,X),
   Padre \== Madre,
   genitore(Madre,Y).

fratelloUnilaterale(X,Y):-
   genitore(GenitoreComune,X),
   genitore(GenitoreComune,Y),
   X \== Y,
   genitore(AltroGenX,X),
   genitore(AltroGenY,Y),
   AltroGenX \== GenitoreComune,
   AltroGenY \== GenitoreComune,
   AltroGenX \== AltroGenY.
```

[1] I "fratellastri"

Notiamo che per individuare i fratelli e' sufficiente che X e Y siano distinti oltre al fatto che Padre e Madre siano distinti.
Nel caso dei fratelli unilaterali la questione e' leggermente piu' complicata in quanto X e Y devono essere distinti. Inoltre X e Y devono avere un genitore comune e un altro genitore(non in comune). I genitori non in comune devono essere chiaramente distinti dal genitore comune e devono anche essere distinti fra loro.

Mostriamo adesso come un ordine intuitivamente corretto delle regole possa portare a risultati disastrosi

Esempio 3 : supponiamo di voler individuare gli antenati all'interno dell'albero genealogico degli esempi precedenti. Definiamo in due modi intuitivamente equivalenti la stessa regola $antenato(X, Y)$

```
%Prima regola
antenato(X,Y):-genitore(X,Y).
antenato(X,Y):-
   genitore(X,Z),
   antenato(Z,Y).

%Seconda regola
antenato(X,Y):-
   antenato(Z,Y),
   genitore(X,Z).
antenato(X,Y):-genitore(X,Y).
```

Queste due regole affermano concettualmente la stessa cosa: affinche' X sia antenato di Y deve essere che X e' genitore di Y oppure che esista un parente Z che li collega. Eppure la prima regola produce i risultati corretti mentre la seconda produce in output:
ERROR: Out of local stack
Questa situazione si verifica poiche' quando l'interprete effettua pattern matching, applica sempre la prima regola disponibile, finendo inevitabilmente in un loop in cui continua ad invocare la regola antenato ed esaurendo lo spazio disponibile nello stack.

23/02/2017

Per riassumere e formalizzare meglio quanto detto finora possiamo dire che i fatti e le regole contengono

- Atomi: gli atomi possono essere *costanti*(la verza, la pancetta...) oppure numeri.
- Variabili
- Termini composti: ottenuti applicando un *funtore* che colleghi altri termini. Ad esempio $sopra(penna, tavolo)$ e' un termine composto da due costanti atomiche. Notiamo che i termini composti possono a loro volta

essere formati da termini composti. Un esempio potrebbe essere $nato(giovanni, data(torino,'12/07/1995'))$

Ad ogni modo possiamo generalizzare i fatti e le regole dicendo che sono entrambi *clausole*. Dunque le clausole possono essere fatti oppure regole.
Quindi in questo senso un programma Prolog e' un'insieme di clausole

1.3 Le liste

Le lista sono la struttura dati piu' utilizzata in Prolog. Una differenza tra Prolog e molti altri linguaggi e' che le liste in Prolog non devono essere necessariamente omogenee

Esempio 1 : la lista $[1, ciao, nato(giovanni, torino), 2]$ e' una normale lista Prolog non omogenea

Un particolare tipo di lista e' la lista vuota indicata con il simbolo [ ].
Per riferirci agli elementi della lista considereremo che la lista sia formata da due parti:

1. La testa, ovvero il primo elemento della lista
2. La coda, ovvero tutti gli altri elementi appartenenti alla lista

Per richiamare queste due parti useremo la notazione $[Head|Tail]$. Vediamone il funzionamento tramite un esempio

Esempio 2 : consideriamo la seguente interrogazione:
$? - [1, 2, 3, 4, 5] = [Head|Tail].$
$Head = 1,$
$Tail = [2, 3, 4, 5]$

Esempio 3 : possiamo anche iterare la suddivisione come segue:
$? - [1, 2, 3, 4, 5] = [Head|[X|Resto]].$
$Head = 1,$
$X = 2,$
$Resto = [3, 4, 5].$

Proseguiamo cercando di fare un po' di pratica con le liste. Nel esempio seguente introdurremo la sintassi dell'assegnamento e faremo vedere come opera la ricorsione.

Esempio 4 : supponiamo di voler scrivere una regola Prolog che calcoli la somma di tutti gli elementi di una lista. Il codice del programma e':

```
somma([ ],0).
somma([Head|Tail], Sum):-
   somma(Tail, SommaTail),
   Sum is Head+SommaTail.
```

Ci sono diverse cose da notare in questo programma. La prima e' che, come avevamo gia' detto, Prolog e' pensato per operare ricorsivamente. Per questo motivo andiamo a definire un caso base e un caso ricorsivo. Il caso base e' banale. Intuitivamente il caso ricorsivo va invece a calcolare la funzione somma sulla parte restante della lista escludendo la testa. Operando ricorsivamente la regola accorcia la lista ad ogni passo, fino a raggiungere il caso della lista vuota, del quale si occupera' il caso base.
Abbiamo anche introdotto la creazione di una variabile che assumera' un certo valore dato dal risultato dell'operazione Head+SommaTail. Di fatto abbiamo introdotto l'operazione di assegnamento.
Notiamo che il punto di vista rispetto ai linguaggi imperativi e' molto diverso. Qui non abbiamo un valore da restituire!
Per poter verificare l'efficacia di questa regola effettuiamo la query:
$? - somma([1,2,3], Risultato)$
$Risultato = 6.$
Come al solito abbiamo creato una variabile(Risultato) sulla quale l'interprete cerchera' di trovare un match per la lista $[1,2,3]$.
Puo' essere interessante andare ad eseguire la query in modalita' trace per studiarne il percorso.

```
?- somma([1,2,3], Risultato).
   Call: (8) somma([1, 2, 3], _602) ? creep
   Call: (9) somma([2, 3], _842) ? creep
   Call: (10) somma([3], _842) ? creep
   Call: (11) somma([], _842) ? creep
   Exit: (11) somma([], 0) ? creep
   Call: (11) _846 is 3+0 ? creep
   Exit: (11) 3 is 3+0 ? creep
   Exit: (10) somma([3], 3) ? creep
   Call: (10) _852 is 2+3 ? creep
   Exit: (10) 5 is 2+3 ? creep
   Exit: (9) somma([2, 3], 5) ? creep
   Call: (9) _602 is 1+5 ? creep
   Exit: (9) 6 is 1+5 ? creep
   Exit: (8) somma([1, 2, 3], 6) ? creep
Risultato = 6.
```

Notiamo che nella prima fase abbiamo una serie di call che ricorsivamente abbreviano la lista. Quando la lista diviene vuota il caso base interviene segnalando il valore zero. A questo punto il valore si propaga alle precedenti chiamate della funzione somma(questo fatto risulta evidente dalle numerose Exit) fino a quando non viene restituito il risultato.

Notiamo che scrivere la regola come segue e' invece pesantemente **sbagliato**

```
somma([ ],0).
```

```
somma([Head|Tail], SommaTail+Head):-
   somma(Tail, SommaTail),
```

Sebbene la tentazione possa essere forte, questo e' un approccio di tipo imperativo che qui non ha alcun senso poiche' l'argomento SommaTail+Head e' soltanto un simbolo e non un'operazione matematica.

Esempio 5 : supponiamo di voler creare una regola che a partire da una lista, vada a creare una nuova lista i cui elementi sono stati tutti incrementati di un certo Delta. Il codice e' il seguente:

```
aggiungiDelta([ ], Delta, [ ]).
aggiungiDelta([Head|Tail], Delta, [NuovaHead|TailAumentata]) :-
   aggiungiDelta(Tail, Delta, TailAumentata),
   NuovaHead is Head+Delta
```

Anche in questo caso abbiamo definito un caso base per la lista vuota e un caso ricorsivo. Andandolo pero' a caricare nell'interprete Prolog riceviamo il seguente warning:

Warning: c:/users/paolo/desktop/somma.pl:11:
Singleton variables: [Delta]

Questo warning si manifesta perche' nel caso base usiamo una variabile(Delta) che non viene mai riferita. Per rimuovere questo warning e' sufficiente sostituire nel caso base "Delta" con "_" che rappresenta le variabili anonime.
Un altro modo(meno elegante) di fare poteva essere:

```
aggiungiDelta([], Delta, []).
aggiungiDelta([Head|Tail], Delta, Risultato) :-
   aggiungiDelta(Tail, Delta, TailAumentata),
   NuovaHead is Head+Delta,
   Risultato = [NuovaHead|TailAumentata].
```

In questo codice abbiamo introdotto il simbolo di uguaglianza che fondamentalmente "forza" la variabile Risultato ad assumere il valore della lista [NuovaHead|TailAumentata], per poi restituirlo come risultato. Per questo motivo e' meno elegante.

27/02/2017

Proseguiamo dando qualche altro esempio di utilizzo delle liste.

Esempio 6 : supponiamo di voler contare gli elementi positivi all'interno di una lista. In questo caso la difficolta' sta nel capire come si possa implementare un costrutto del tipo if-then-else. La soluzione consiste nell'implementare due regole, una nel caso l'elemento corrente sia positivo, e un'altra se l'elemento corrente e' negativo

```
positivi([], 0).
positivi([Head|Tail], Somma) :-
   Head > 0,
   positivi(Tail, SommaParz),
   Somma is SommaParz+1.
positivi([Head|Tail], Somma) :-
   Head =< 0,
   positivi(Tail, SommaParz),
   Somma is SommaParz.
```

Potremmo chiederci se allora il numero di regole da introdurre all'aumentare delle condizioni cresce secondo la potenza di due. Per ora purtroppo si', ma in seguito vedremo dei metodi per evitare il proliferare delle regole

Esempio 7 : Come possiamo invece utilizzare il Prolog per invertire una lista? L'unica "funzione" predefinita e' la append[2]:

```
invertiLista([],[]).
invertiLista([Head|Tail], Result) :-
   invertiLista(Tail, ListaParz),
   append(ListaParz, [Head], Result).
```

Eseguendo questo programma in modalita' trace purtroppo ci accorgiamo che per come e' implementata la append, ogni volta che viene invocata scorre l'intera lista. Questo porta ad un costo di inversione quadratico nel numero degli elementi della lista. Possiamo dare una versione ottimizzata, che abbia invece costo lineare:

```
invertiOpt([], Temp, Temp).
invertiOpt([Head|Tail], Temp, Result) :-
   invertiOpt(Tail, [Head|Temp], Result).
```

La versione ottimizzata e' decisamente meno intuitiva per come e' scritta ma piu' efficiente. Quello che fa e' appoggiarsi a una variabile temporanea Temp ed eseguire l'inversione man mano che "scende" nelle varie chiamate ricorsive di invertiOpt.
Notiamo infatti che e' diverso dal solito anche il caso base. Quando la lista e' vuota, diciamo che il risultato finale(il terzo argomento) coincide con il cio' che e' contenuto nella variabile temporanea.
Infine, se volessimo dare una regola che sfrutti invertiOpt ma che abbia una segnatura uguale a quella della regola invertiLista, sarebbe sufficiente scrivere una nuova regola:

```
invertiListaOpt(Lista, ListaInv):-
   invertiOpt(Lista, Temp, ListaInv).
```

[2]Avente la ben nota semantica di inserire in coda gli elementi

2 Interprete Prolog e risoluzione SLD

2.1 Logica classica e risoluzione SLD

A questo punto possiamo analizzare in dettagli come avviene l'esecuzione dei programmi in Prolog. L'esecuzione e' detta a *backward chaining* poiche' parte dal goal. L'obiettivo e' infatti dimostrare che il goal sia conseguenza logica delle clausole. Per farlo usiamo le regole.
Il programma del'interprete Prolog e' il seguente

```
interprete(G)
   if G vuoto
      successo
   else
      sia G=G1, G2, .., Gn
      sia R insieme delle regole applicabili a G1
      if R vuoto
         fallimento
      else
         scegliere una regola A :- B1, B2, .., Bm in R
         sia s il MGU di G1 e A
         interprete (B1s, B2s, .., Bms, G2s, .., Gns)
```

Quello che nel programma interprete viene indicato con s(o piu' spesso con σ) e' il *most general unifier*, o l'*unificatore generale*(MGU). Ovvero e' il minimo insieme delle sostituzioni che servono per "unificare"(vedremo questo procedimento piu' avanti) due clausole.
L'idea e' che esistano un'insieme di clausole che vengono verificate. Quando l'insieme delle clausole viene svuotato, allora abbiamo raggiunto il goal(successo). In generale quindi, l'interprete cerca di verificare il goal applicando delle regole che ci permettano di raggiungerlo.
Notiamo che di per se' l'interprete non sarebbe deterministico visto che sceglie **una** regola, senza specificare quale. Il determinismo ci viene fornito nel momento in cui specifichiamo che la regola da applicare e' la prima che troviamo nel programma.
Come leghiamo pero' la logica al nostro interprete Prolog? Partiamo dalle basi, ovvero dalla logica classica.

Nella logica classica supponiamo di voler rappresentare una certa base di conoscenza K che contiene le seguenti affermazioni:
$gatto \rightarrow miagola$
$persiano \rightarrow gatto$

Possiamo affermare che $persiano \rightarrow miagola$ e' conseguenza logica della base di conoscenza? Ovvero possiamo dire che

$$K \models (persiano \rightarrow miagola)$$

Per dimostrarlo potremmo costruire una serie di tabelle di verita'. Prima le tabelle di $gatto \rightarrow miagola$, $persiano \rightarrow gatto$. Poi la tabella di verita' di $(gatto \rightarrow miagola) \wedge (persiano \rightarrow gatto)$. Infine potremmo costruire la tabella di verita' di $persiano \rightarrow miagola$.
A questo punto diremo che $K \models (persiano \rightarrow miagola)$ se ogni volta(ovvero per gli stessi input) che $(gatto \rightarrow miagola) \wedge (persiano \rightarrow gatto)$ e' vera, e' vera anche $persiano \rightarrow miagola$.

Purtroppo con le tabelle di verita' facciamo troppo lavoro! È per questo motivo che usiamo invece un *metodo di prova*.
Il metodo di prova e' una procedura che verifica se una formula e' conseguenza logica della teoria. Il primo metodo di prova che vediamo e' il *metodo di risoluzione*.
Nel metodo di risoluzione rappresentiamo la base di conoscenza in una forma a clausole e abbiamo una sola regola:
supponendo di avere due clausole $C_1 = A_1 \vee \ldots \vee A_n$ e $C_2 = B_1 \vee \ldots \vee B_m$, supponendo inoltre che esistano due letterali A_i e B_j tali per cui $A_i = \neg B_j$ allora ne possiamo derivare una *clausola risolvente*

$$A_1 \vee \ldots A_{i-1} \vee A_{i+1} \vee \ldots \vee A_n \vee B_1 \vee \ldots B_{j-1} \vee B_{j+1} \vee \ldots \vee B_m$$

Dunque la clausola risolvente e' praticamente l'unione delle due clausole da cui pero' rimuoviamo il termine A_i e il termine B_j.
Partendo dalla base di conoscenza a cui aggiungiamo il goal negato, l'obiettivo del metodo di risoluzione e' trovare una inconsistenza, ovvero raggiungere la clausola vuota[3]. Vediamo un esempio per comprendere meglio come funzioni il metodo di risoluzione

Esempio 1 : supponiamo di voler risolvere nuovamente il problema che abbiamo visto poco fa riguardo i gatti. Partendo dalla nostra base di conoscenza, andiamo a negare il goal:

$$\neg(persiano \rightarrow miagola) \equiv \neg(\neg persiano \vee miagola) \equiv \\ (persiano) \wedge (\neg miagola)$$

adesso possiamo aggiungere il goal negato alla base di conoscenza e cercare di trovare una contraddizione. Riordiniamo le clausole e notiamo che per la regola di risoluzione

$$(\neg gatto \vee miagola) \wedge (\neg miagola) \rightarrow (\neg gatto) \\ (\neg persiano \vee gatto) \wedge (persiano) \rightarrow (gatto)$$

[3]Logicamente parlando potremmo cercare di aggiungere il goal e controllare se la base di conoscenza e' verificata in ogni configurazione. Analogamente pero' possiamo anche aggiungere il goal negato e cercare di trovare una contraddizione

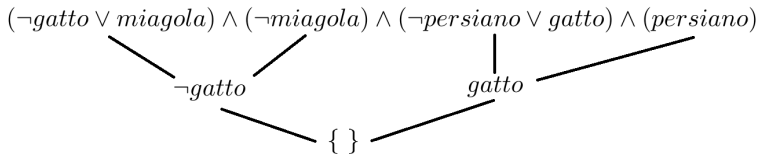

Figura 2: Il procedimento di risoluzione mentre opera sull'esempio 1

Avendo ottenuto le due nuove clausole $(\neg gatto)$ e $gatto$ sempre per la regola di risoluzione ricaviamo che:

$$(\neg gatto) \wedge gatto \rightarrow \{\ \}$$

Dunque abbiamo ottenuto una contraddizione e il goal e' quindi implicato dalla base di conoscenza! L'intero procedimento viene mostrato graficamente in Figura 2.

Potremmo pensare di estendere questo procedimento alla logica del primo ordine. A questo punto, usando le variabili potremmo unificare le varie clausole a patto che l'MGU determini un'insieme di sostituzioni da fare sui nomi.

01/03/2017

Piu' formalmente, nella logica del primo ordine, se le due clausole $C_1 = A_1 \vee \ldots \vee A_n$ e $C_2 = B_1 \vee \ldots \vee B_m$ contengono variabili, possiamo effettuare l'unificazione se esiste una sostituzione σ delle variabili affinche' $A_i = \neg B_j$.
Vediamone subito un esempio per comprenderlo meglio

Esempio 2 : supponiamo che la nostra base di conoscenza sia:

K: $\forall x(Chirurgo(x) \Rightarrow Medico(x))$
$\forall x(Medico(x) \Rightarrow Laureato(x))$
$Chirurgo(figlioDi(franco))$

Possiamo dire che $K \models Laureato(figlioDi(franco))$? Intuitivamente si', cerchiamo di dimostrarlo. Per farlo eliminiamo le implicazioni e poniamo le varie clausole in forma congiuntiva. Il procedimento di dimostrazione e' mostrato in Figura 3

La notazione $x/figlioDi(franco)$ indica che nella clausola e' stata effettuata la sostituzione della variabile con

In generale quando possiamo applicare l'unificazione? La tabella successiva ce lo mostra:

	cost. c_2	var. x_2	composto s_2
cost. c_1	$c_1 = c_2$	x_2/c_1	non unificano
var. x_1	x_1/c_2	x_1/x_2	x_1/s_2
composto s_1	non unificano	x_2/s_1	solo se il funtore in s_1 e s_2 e' lo stesso e gli argomenti unificano

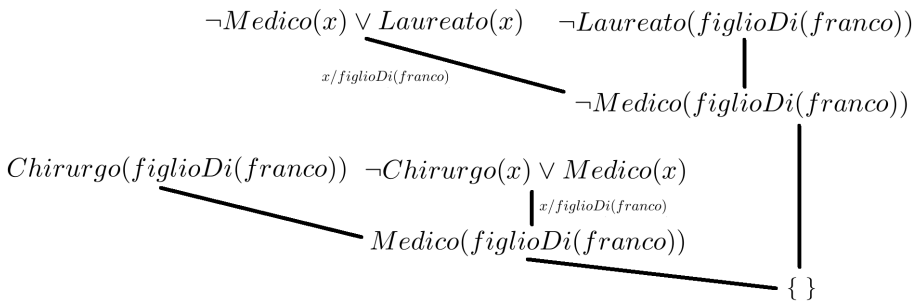

Figura 3: Il procedimento di risoluzione dell'esempio 2

La tattica utilizzata dal Prolog e' simile ma piu' efficiente. Utilizza le clausole in una forma particolare detta *di Horn*. Le clausole di Horn sono clausole che contengono al piu' un letterale non negato. Il tipo di risoluzione del Prolog e' la risoluzione SLD: avendo a disposizione una base di conoscenza K formata da sole clausole di Horn applica una strategia di *linear input*, ovvero ad ogni passo una clausola viene prese dalla K di partenza, mentre l'altra e' sempre la clausola risolvente del passo precedente.
Questa strategia non sarebbe completa[4] in generale, ma lo e' se vengono utilizzate delle clausole di Horn.
Il procedimento di risoluzione puo' concludersi in tre modi differenti:

- Successo: se otteniamo la clausola vuota
- Fallimento finito: se non possiamo derivare il goal o comunque non riusciamo ad ottenere la clausola vuota
- Fallimento infinito: se e' sempre possibile derivare nuove clausole risolventi

Possiamo rappresentare tutte le derivazioni ottenute partendo dal goal con un albero SLD. La radice e' il goal iniziale e ogni nodo dell'albero e' un atomo selezionato a cui viene applicata la sostituzione dell'MGU.
Come gia' accaduto precedentemente, di per se' non avremmo un determinismo visto che con la risoluzione SLD non stabiliamo quale sia il nodo da sviluppare all'interno dell'albero. Affinche' il procedimento sia deterministico, nel Prolog si sviluppa sempre il nodo piu' a sinistra(tale regola viene detta *leftmost*) e le clausole vengono considerate nell'ordine in cui sono scritte nel programma. La strategia di ricerca e' in profondita' con backtracking.
Questa strategia non e' completa poiche' se una computazione che porta al successo si trova a destra di un ramo infinito, l'interprete non lo trova perché entra, senza mai uscirne, nel ramo infinito.

06/03/2017

cerchiamo di capire meglio come il Prolog crea e sfrutta l'albero SLD. Per farlo

[4]Ricordiamo che un algoritmo viene detto corretto se trova soluzioni giuste e completo se trova tutte le soluzioni giuste

ricordiamo anzitutto che il metodo di risoluzione visto e' corretto e completo solo per clausole Horn. Consideriamo i vari elementi di un programma Prolog:

- Fatti: essendo clausole unarie sono per forza di cose clausole Horn
- Regole: consideriamo una generica regola
  $A :- B_1, \ldots, B_m$
  che possiamo scrivere come
  $(B_1 \wedge \ldots \wedge B_m) \Rightarrow A$.
  Dalle leggi di De Morgan sappiamo che
  $(B_1 \wedge \ldots \wedge B_m) \Rightarrow A \equiv \neg B_1 \vee \ldots \vee \neg B_m \vee A$
  Anche le regole sono dunque clausole di Horn visto che l'unico elemento non negato e' A. Questo ha pero' una conseguenza importante: nessuna parte di $B_1, \ldots, B_m$ puo' essere introdotta negativamente. Altrimenti quando viene risolta l'implicazione non otterremmo una clausola di Horn
- Goal: consideriamo un goal generico formato da piu' parti
  $G_1, \ldots, G_n \equiv G_1 \wedge \ldots \wedge G_n$
  Ricordando che in SLD operiamo per refutazione, il goal negato diventa
  $\neg(G_1 \wedge \ldots \wedge G_n) \equiv \neg G_1 \vee \ldots \vee G_n$
  Anche il goal e' una clausola di Horn. Anche in questo caso ricaviamo un dato importante: Prolog non permette di specificare un goal con elementi negati, altrimenti anche in questo caso potremmo non ottenere delle clausole di Horn

A questo punto siamo convinti che tutti gli elementi di un programma Prolog sono delle clausole di Horn. Dunque il procedimento di risoluzione sara' corretto e completo.
Vediamo come opera Prolog con un esempio

Esempio 3 : consideriamo di avere a disposizione il seguente insieme di clausole di un programma Prolog che rappresenta un albero genealogico:

```
genitore(mario, carlo).          %clausola 1
genitore(franco, mario).         %clausola 2
genitore(luca, franco).          %clausola 3
antenato(X,Y):-genitore(X,Y).    %clausola 4
antenato(X,Y):-                  %clausola 5
   genitore(X,Z),
   antenato(Z,Y).
```

Mostriamo in Figura 4 l'albero SLD.

Notiamo che abbiamo dovuto convertire le clausole Prolog in clausole della logica classica. Su ogni ramo sono rappresentate le istanziazioni delle variabili e il numero della regola che e' stata usata.
Inoltre notiamo che abbiamo costruito l'albero di risoluzione SLD e abbiamo applicato le regole come avrebbe fatto il Prolog, ovvero seguendo la politica leftmost con backtracking

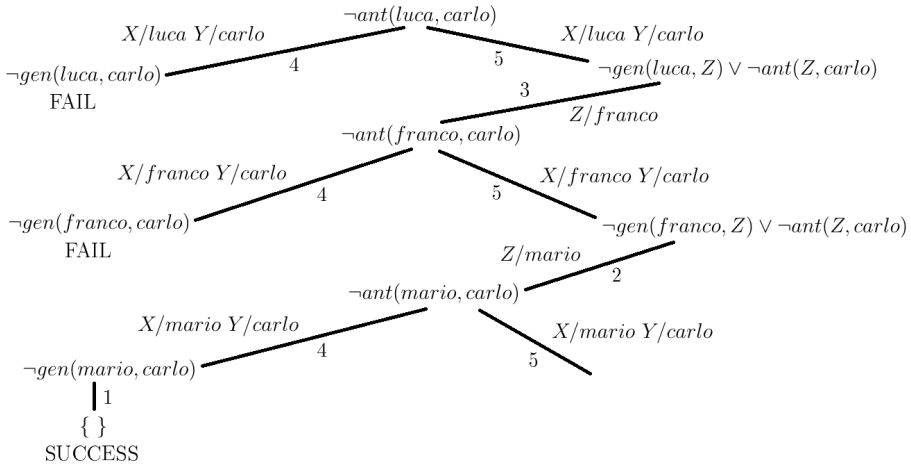

Figura 4: Il procedimento di risoluzione dell'esempio 3

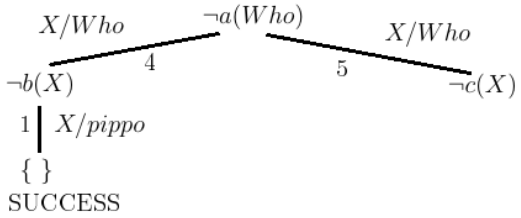

Figura 5: Il procedimento di risoluzione dell'esempio 4

Esempio 4 : consideriamo il seguente programma Prolog:

```
b(pippo).              %clausola 1
d(gino).               %clausola 2
e(gino).               %clausola 3
a(X) :- b(X).          %clausola 4
a(X) :- c(X).          %clausola 5
c(X) :- d(X), a(X).    %clausola 6
c(X) :- e(X).          %clausola 7
```

Supponiamo di voler fare come gia' altre volte abbiamo fatto. Ovvero di inserire nel goal una variabile. In questo caso l'albero SLD dell'esecuzione della regola $a(Who)$ viene mostrato in Figura 5

Consideriamo invece un caso un po' piu' particolare. Se il goal fosse $a(gino)$? Il risultato e' mostrato in Figura 6

Notiamo che in questo caso il problema e' che ad un certo punto dell'albero incontriamo nuovamente il nodo radice! Questo significa che il nostro

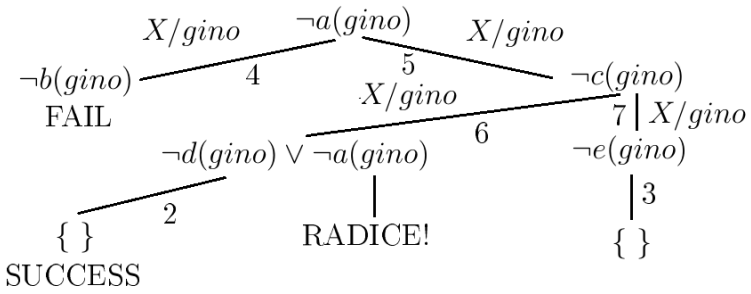

Figura 6: Il procedimento di risoluzione dell'esempio 4

programma Prolog andra' in loop continuando ad esplorare un sottoalbero infinito.

2.2 Procedimento di unificazione

A questo punto possiamo parlare maggiormente nel dettaglio del processo di unificazione in Prolog. Intuitivamente unificare due clausole significa trovare una sostituzione delle variabili all'interno delle clausole affinche' dopo la sostituzione, tali clausole risultino uguali.
La *sostituzione* σ viene denotata come segue

$$\sigma = \{x_1/t_1, \ldots, x_n/t_n\}$$

Vediamo un esempio

Esempio 1 : dato il termine $t = f(x_1, g(x_2), a)$ e la sostituzione $\sigma\{x_1/h(x_2), x_2/b\}$ abbiamo che $t\sigma = f(h(x_2), g(b), a)$. Notiamo subito che le istruzioni sono "istantanee" anche se abbiamo dato una sostituzione per x_2, non lo andiamo a sostituire in $h(x_2)$. Questo perche' la sostituzione viene fatta una sola volta e non "ricorsivamente".

Ovviamente e' sempre valida la composizione di sostituzioni

Esempio 2 : supponiamo di riprendere il termine dell'esempio precedente. Applichiamo due volte la sostituzione σ. Otteniamo:
$t\sigma\sigma = f(h(b), g(b), a)$

A questo punto possiamo dire che due termini t_1, t_2 sono *unificabili* se esiste una sostituzione σ(detta unificatore) che li rende identici: $t_1\sigma = t_2\sigma$
Molto spesso capita che esista piu' di una sostituzione che porti ad unificare i due termini. In quel caso quale sostituzione e' meglio usare? Si decide di utilizzare la sostituzione *piu' generale*.
Una sostituzione θ e' piu' genera di una sostituzione σ se esiste una sostituzione λ tale per cui:

$$\sigma = \theta\lambda$$

Esempio 3 : supponiamo di avere le tre seguenti sostituzioni:
$\sigma_1 = \{x1/g(x3), x2/x3, x4/h(g(x3))\}$
$\sigma_2 = \{x1/g(a), x2/a, x3/a, x4/h(g(a))\}$
$\lambda = \{x3/a\}$
Allora la sostituzione σ_1 e' piu' generale della sostituzione σ_2 perche'
$\sigma_2 = \sigma_1\lambda$

In generale e' possibile dimostrare che se due termini sono unificabili esiste sempre anche un *unificatore piu' generale* anche detto MGU(Most General Unifier).

07/03/2017

Vediamo adesso *l'algoritmo di Martelli-Montanari* per determinare l'MGU. Tale algoritmo si basa fondamentalmente su due regole

1. Term reduction: sia $f(t_1^{'}, \ldots, t_n^{'}) = f(t_1^{''}, \ldots, t_n^{''})$ una equazione dell'insieme. Possiamo ottenere un nuovo insieme andando a sostituirla con $t_1^{'} = t_1^{''}, \ldots, t_n^{'} = t_n^{''}$

2. Variable elimination: sia $x = t$ un'equazione dell'insieme dove x e' una variabile mentre t e' un termine. Possiamo ottenere un nuovo insieme di equazioni applicando la sostituzione $\theta = \{x/t\}$ a tutte le altre equazioni.

Vediamo adesso le regole dell'algoritmo.
Dato un insieme di equazioni, l'algoritmo procede eseguendo ripetutamente le seguenti trasformazioni. L'algoritmo termina o con fallimento o con successo

- Scegliere una equazione della forma t = x dove t non e' una variabile e x lo e', e sostituirla con x = t.

- Scegliere una equazione della forma x = x e cancellarla.

- Scegliere una equazione della forma $t^{'} = t^{''}$ dove $t^{'}$ e $t^{''}$ non sono variabili. Se i due simboli di funzione di $t^{'}$ e $t^{''}$ sono diversi, fallimento. Altrimenti applicare term reduction.

- Scegliere una equazione della forma x = t dove x e' una variabile che occorre in qualche altra parte nell'insieme di equazioni e dove $t \neq x$. Se x occorre in t, fallimento. Altrimenti applicare variable elimination.

Applichiamo l'algoritmo ad un esempio

Esempio 4 : supponiamo di avere il seguente insieme di equazioni.
$\{g(x_2) = x_1, f(x_1, h(x_1), x_2) = f(g(x_3), x_4, x_3)\}$
Applichiamo term reduction alla seconda equazione
$\{g(x_2) = x_1, x_1 = g(x_3), h(x_1) = x_4, x_2 = x_3\}$
Variable elimination alla seconda equazione
$\{g(x_2) = g(x_3), x_1 = g(x_3), h(g(x_3)) = x_4, x_2 = x_3\}$
Term reduction alla prima equazione
$\{x_2 = x_3, x_1 = g(x_3), x_4 = h(g(x_3)), x_2 = x_3\}$

Variable elimination alla prima equazione
$\{x_2 = x_3, x_1 = g(x_3), x_4 = h(g(x_3))\}$
L'MGU sara' quindi
$\theta = \{x_1/g(x_3), x_2/x_3, x_4/h(g(x_3))\}$

3 Prolog

3.1 Comando cut

Avevamo visto come possa essere in certi casi problematico implementare il costrutto if-then-else in quanto avendo n condizioni dobbiamo implementare 2^n casi. Introduciamo il comando cut che in qualche modo implementa proprio l'if-then-else. Vediamo un esempio

Esempio 1 : supponiamo di voler realizzare un programma che conta le occorrenze di un numero in una lista. Il codice che avremmo dato finora era

```
conta(_,[],0).
conta(X, [X|Tail], N):-
   conta(X,Tail, N1),
   N is N1+1.
conta(X, [Head|Tail], N):-
   conta(X, Tail, N).
```

Questo codice ha il grave problema che non blocca il backtracking del Prolog. Infatti quando viene trovata la prima soluzione giusta, l'interprete Prolog risale nell'albero SLD per cercare altri casi di match, e in effetti ne trova uno per ogni elemento della lista, perche' per ogni elemento possiamo applicare due regole distinte(la prima o la seconda). Consideriamo invece il seguente codice

```
conta(_,[],0).
conta(X, [X|Tail], N):- !,
   conta(X,Tail, N1),
   N is N1+1.
conta(X, [Head|Tail], N):-
   conta(X, Tail, N).
```

Il simbolo !- posto dopo il funtore della prima regola indica che quando Prolog passa da quel punto del programma, se in seguito vi sara' backtracking sulla regola che contiene il cut, il backtracking verra' stoppato

Quello che dunque fa il comando cut e' seguire un solo ramo, implementando di fatto l'if-then-else. Vediamo subito un altro esempio

Esempio 2 : supponiamo di voler implementare il programma Prolog che effettua il merge di due liste ordinate. La versione senza cut sarebbe

```
merge(X, [], X).
merge([], X, X).
merge([X|Tail1], [X|Tail2], [X,X| MergeTail]):-
   merge(Tail1, Tail2, MergeTail).
merge([H1|Tail1], [H2|Tail2], [H1|MergeTail]):-
   H1<H2,
   merge(Tail1, [H2|Tail2], MergeTail).
merge([H1|Tail1], [H2|Tail2], [H2|MergeTail]):-
   H2<H1,
   merge([H1|Tail1], Tail2, MergeTail).
```

Invece possiamo usare cut per evitare tutte le operazioni di backtracking inutili

```
mergeOpt(X, [], X):-!.
mergeOpt([], X, X):-!.
mergeOpt([X|Tail1], [X|Tail2], [X,X| MergeTail]):-
   mergeOpt(Tail1, Tail2, MergeTail).
mergeOpt([H1|Tail1], [H2|Tail2], [H1|MergeTail]):-
   H1<H2,
   !,
   mergeOpt(Tail1, [H2|Tail2], MergeTail).
mergeOpt([H1|Tail1], [H2|Tail2], [H2|MergeTail]):-
   mergeOpt([H1|Tail1], Tail2, MergeTail).
```

Notiamo due cose: la prima e' che abbiamo convertito i due fatti(i casi base) in due regole. Quello che stiamo dicendo e': se raggiungi il caso in cui almeno una delle due liste e' vuota, fermati senza fare alcun backtracking. Il secondo fatto interessante e' che abbiamo rimosso dall'ultima regola di mergeOpt il controllo su quale dei due elementi in testa sia maggiore dell'altro. Questo perche' il secondo caso verra' preso soltanto se non viene presa la regola precedente, ovviamente a causa del cut!

08/03/2017

Proseguiamo facendo ancora un paio di esempi prima di vedere piu' formalmente il cut

Esempio 3 : nel linguaggio Prolog esiste il costrutto $member(X, Lista)$ che verifica se X appartenga a *Lista*. Proviamo a implementarlo noi dando due versioni diverse, una che usa il cut, una che non lo usa

```
%appartiene(X|L).
appartiene(X,[X|_]).
appartiene(X,[_|Tail]):-appartiene(X,Tail).

appartieneWrong(X,[X|_]):-!.
appartieneWrong(X,[_|Tail]):-appartieneWrong(X,Tail).
```

La prima implementazione e' corretta. La seconda invece no! Infatti nella prima l'invocazione di "appartiene" si comporta come si comporterebbe la member. Mentre nel secondo caso restituisce solo il primo risultato

Esempio 4 : supponiamo di voler unire(unione insiemistica)due liste. Gli elementi all'interno di ogni lista non possono essere ripetuti ma possono esserci alcuni elementi in comune tra la prima e la seconda lista. In tal caso la lista risultato non dovra' contenere la doppia occorrenza. Il codice che implementa l'unione e':

```
unione([],L2,L2):-!.
unione(L1,[],L1):-!.
unione([X|Tail],L2,ListaUnione):-
   member(X,L2),
   !,
   unione(Tail,L2,ListaUnione).
unione([X|Tail],L2,[X|ListaUnione]):-
   unione(Tail,L2,ListaUnione).
```

Senza utilizzare il comando cut sarebbe stato impossibile realizzare tale unione.

3.2 Controllo del programma

Ora che abbiamo visto l'importanza del cut in vari esempi, cerchiamo di darne una definizione formale. Per comprenderlo andiamo a studiare il modello a runtime di prolog.
Fondamentalmente sfrutta due strutture a stack:

1. Stack di esecuzione: contiene le clausole su cui stiamo operando. In posizione top contiene la clausola su cui stiamo operando in questo momento
2. Stack di backtracking: tiene traccia delle possibili regole applicabili per gli elementi contenuti nello stack. In posizione top contiene le regole applicabili per l'elemento posto al top dello stack di esecuzione.

Cerchiamo di capire meglio come operano questi due stack[5] tramite un esempio.

Esempio 1 : supponiamo di avere il seguente programma Prolog

```
a :- p,b.  %cl1
a :- p,c.  %cl2
p.         %cl3
```

Supponiamo di avviare la query
$? - a.$

Mostriamo in Figura 7 l'evoluzione dello stack nel corso della query

[5] Anche se a livello implementativo del Prolog si sviluppano entrambi su uno stack unico

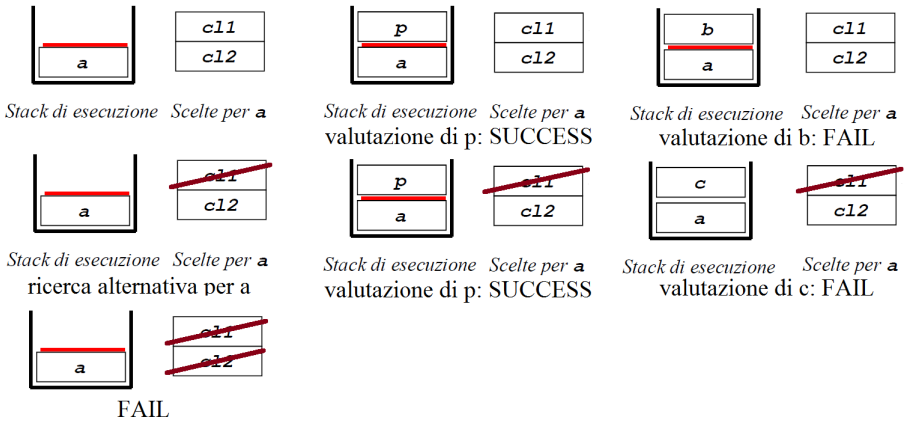

Figura 7: Configurazioni dei due stack nei vari passaggi dell'esecuzione dell'esempio 1

Quello che succede e' che ad ogni passo consideriamo le alternative per l'elemento sul top dello stack. Quando abbiamo un successo o un fallimento rimuoviamo l'elemento posto al top dello stack.

L'utilizzo del cut fondamentalmente rende definitive le scelte fatte fino a un certo punto andando a rimuovere alcuni elementi dallo stack di backtracking. Formalmente, data una clausola

$$p :- q_1, \ldots, q_i, !, q_{i+1}, \ldots, q_n$$

nel momento in cui l'interprete Prolog incontra il simbolo di cut vengono eliminate dallo stack di backtracking tutte le scelte per le clausole $p, q_1, \ldots, q_i$.
Applichiamo subito il meccanismo in un esempio

Esempio 2 : supponiamo di avere il seguente programma Prolog

```
g :- a.        %cl1
g :- s.        %cl2
a :- p,!,b.    %cl3
a :- r.        %cl4
p :- q.        %cl5
p :- r.        %cl6
r.             %cl7
```

E supponiamo di voler avviare la seguente query:
$? - g$
Intuitivamente possiamo vedere che il goal dovrebbe essere raggiunto visto che $r \Rightarrow a \Rightarrow g$ e r e' un fatto a nostra disposizione.
Invece non lo raggiungeremo proprio a causa del cut.
Mostriamo in Figura 8 l'evoluzione dello stack

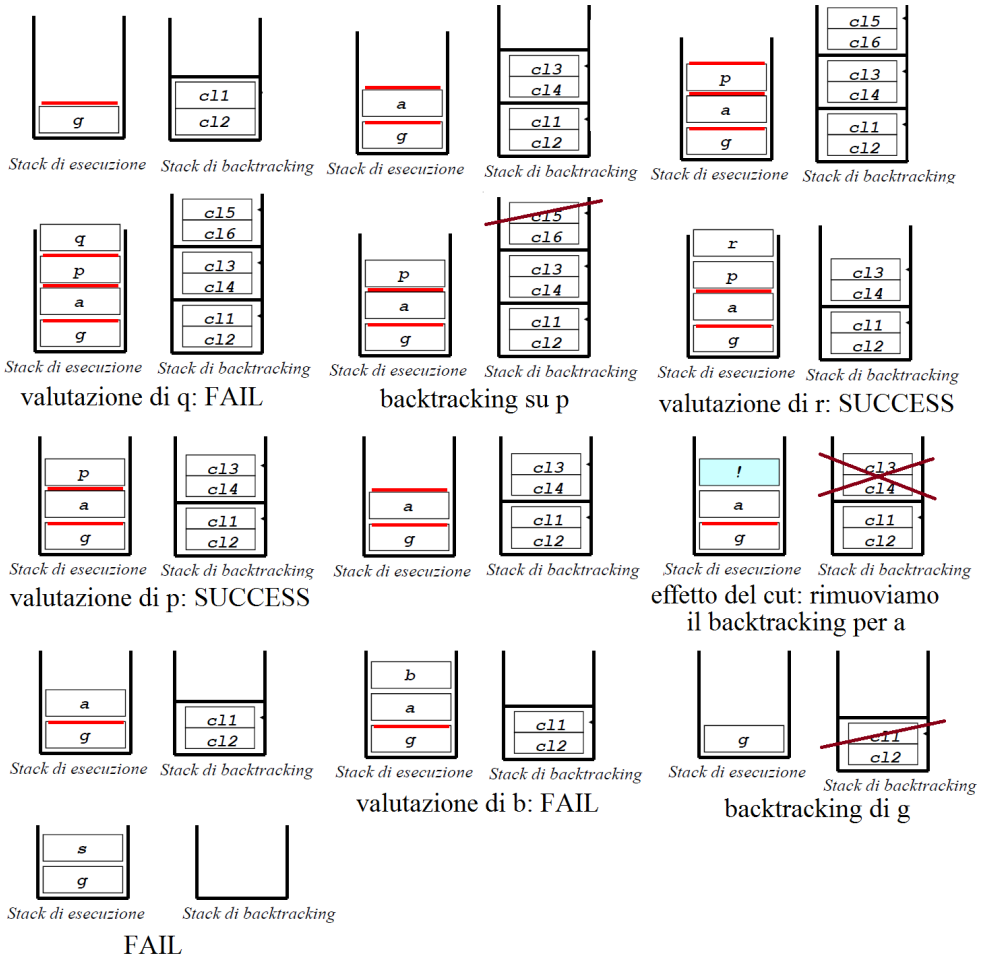

Figura 8: Configurazioni dei due stack nei vari passaggi dell'esecuzione dell'esempio 2

Notiamo che andiamo ad effettuare il backtracking fino a quando non incontriamo il comando cut.
A quel punto, eliminiamo dallo stack del backtracking tutte le alternative per p(che non ne ha di allocate in quel momento) e di a(che invece ne ha, cl3 e cl4) vengono rimosse dallo stack di backtracking.
Notiamo quindi che sebbene ci aspettassimo di ottenere un successo questo non avviene poiche' il cut cambia radicalmente il significato di un programma, andando ad eliminare tutte le possibili alternative per quei termini che si collocano prima di esso nella stessa regola.
Questo non e' necessariamente un male perche' ad esempio quel cut poteva stare a significare

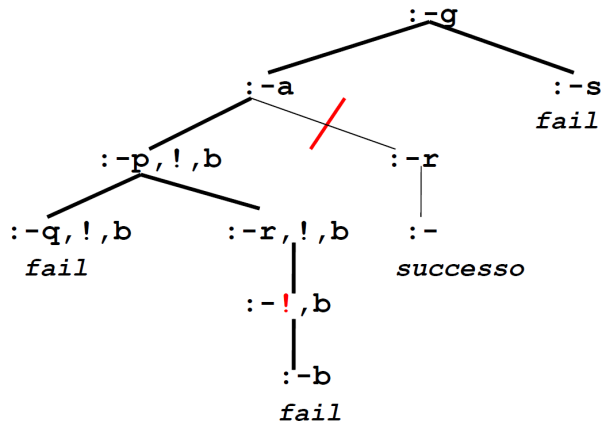

Figura 9: Albero SLD dell'esempio 2

```
if(p) then b
else r
```

e magari era proprio questo il significato che volevamo dare al nostro programma.
Un'altra rappresentazione che possiamo dare del funzionamento del cut sta nell'albero SLD che mostriamo in Figura 9

Cut elimina il ramo di a che altrimenti avremmo potuto prendere.

09/03/2017

Vediamo un altro esempio che sfrutta il cut

Esempio 2 : supponiamo di voler implementare l'intersezione tra due liste. Il codice e' il seguente

```
intersezione([],_,[]):-!.
intersezione(_,[],[]):-!.
intersezione([X|Tail1], L2, [X|IntersezioneCoda]):-
   member(X, L2),!,
   intersezione(Tail1,L2,IntersezioneCoda).

intersezione([_|Tail1], L2, IntersezioneCoda):-
   intersezione(Tail1, L2, IntersezioneCoda).
```

In Prolog possiamo anche specificare se gli argomenti di una regola siano un input un output oppure possano essere sia input che output. Gli argomenti che saranno sempre un input vengono preceduti dal simbolo +, quelli che sono sempre un output sono preceduti dal simbolo − mentre quelli che possono essere sia input che output vengono preceduti dal simbolo ?

Esempio 3 : supponiamo di voler implementare una regola che data una lista e una posizione restituisce l'elemento della lista in quella posizione. La regola puo' essere scritta come
$nth(+lista,\ +p,\ -elem)$
Supponiamo invece di voler realizzare una regola simile ma che restituisca l'elemento nel caso in cui venga data la posizione, e viceversa che possa restituire la posizione se viene dato l'elemento. In questo caso scriveremmo:
$nth(+lista,\ ?p,\ ?elem)$

3.3 Negazione per fallimento

Affrontiamo ora un argomento abbastanza semplice ma importante. In Prolog abbiamo detto che non possiamo specificare l'operatore logico *not*. Questo per fare in modo di avere sempre clausole horn.
In realta' nel tempo sono stati introdotti dei *predicati extralogici* che permettono di specificarlo. In particolare la sintassi del not e' \+. Prima di proseguire applichiamolo subito con un esempio

Esempio 1 : supponiamo di voler stabilire un criterio per definire un *campione*(in termini calcistici). Diremo allora che un campione e' colui che e' un top player e che non sfotte i suoi avversari. Il codice che usiamo e' il seguente

```
campione(X):-
   topPlayer(X),
   \+sfotte(X).
topPlayer(neymar).
topPlayer(messi).
sfotte(neymar).
```

Se in questo caso andiamo ad eseguire la query:
$?-campione(neymar).$
$false.$
la risposta e' come ci aspettiamo $false$. Se invece provassimo ad eseguire la query:
$?-campione(messi).$
$true.$

Cerchiamo di comprendere meglio i due risultati ottenuti nell'esempio precedente. Il primo e' quello che ci aspettavamo, il secondo invece potrebbe sorprenderci.

L'idea nella negazione per fallimento e' che quando si presenta una negazione, Prolog cerca di dimostrare invece l'affermazione. Ovvero, se l'interprete Prolog incontra $\backslash + A$, cerca di dimostrare A. Nel caso in cui riesca a dimostrare A allora l'interprete restituira' $false$, altrimenti restituira' $true$.
Questo spiega i risultati ottenuti nell'esempio precedente. Quando veniva chiesto

$?-campione(neymar)$. l'interprete incontrando la clausola $\backslash + sfotte(neymar)$ cercava di dimostrare $sfotte(neymar)$.
Avendo trovato tale fatto nella base di conoscenza la dimostrazione di $sfotte(neymar)$ restituisce $true$ e dunque la sua negazione restituisce $false$. Questo spiega anche il risultato per $campione(messi)$.. Infatti l'interprete cerca di dimostrare $sfotte(messi)$ e non trovando tale fatto nella base di conoscenza $sfotte(messi)$ restituisce $false$. E dunque la sua negazione restituisce $true$. In tal modo messi risulta essere un campione.

Esempio 2 : supponiamo di avere il seguente programma Prolog

```
p(a).
p(b).
q(a).
r(X):-p(X),q(X).
```

Supponendo di eseguire la seguente query
$:-true, \backslash + r(a)$.
$false$. Come ci aspettiamo la query ha restituito $false$. Ne approfittiamo per specificare che per motivi di implementazione dell'interprete Prolog, la negazione non puo' essere inserita in prima posizione in una regola. Dunque se abbiamo regole ad una sola clausola negata(come in questo caso) e' sufficiente aggiungere una clausola fittizia $true$ prima della clausola da negare

Esempio 3 : supponiamo invece di avere il seguente programma Prolog

```
p(a).
p(b).
q(a).
s(X):-p(X),t(X).
t(X):-q(X).
t(X):-p(X),s(X).
```

supponiamo di eseguire la seguente query
$:-true, \backslash + s(b)$.
In questo caso la verifica di $s(b)$ porta ad un loop, e quindi l'interprete Prolog non termina. Questo e' un tipico caso di *negazione senza esito*.

Riprendiamo adesso l'esempio 1.

Esempio 4 : supponiamo di modificare il programma aggiungendo il fatto $sfotte(messi)$. Andando ad eseguire la query:
$?-campione(messi)$.
La risposta che riceviamo e' ovviamente $false$, come ci aspettiamo.
Dobbiamo pero' notare una cosa importante. Abbiamo abbandonato il ragionamento *monotono* tipico della logica classica.

Nella logica classica sappiamo che se:
$K \models F$ allora varra' anche che $K \cup \{C\} \models F$
Nell'esempio precedente questo non si e' verificato infatti aggiungendo un fatto($sfotte(messi)$) non siamo piu' in grado di dimostrare che $campione(messi)$. Questo e' un fatto tipico del ragionamento non monotono. Questo passaggio da logica monotona a non monotona e' ovviamente dovuto all'introduzione del predicato extralogico \+
Mostriamo infine una particolarita' della negazione

Esempio 5 : supponiamo di avere lo stesso programma dell'esempio 1. Andando ad eseguire la query
$: -true, \backslash + campione(chi).$
$false.$
Dunque non otteniamo i nomi di tutti coloro che non sono campioni. Questo si verifica perche' la negazione non restituisce l'istanziazione delle variabili ma va solo a verificare
$campione(chi)$.
Nel momento in cui un campione viene effettivamente trovato(messi) allora la verifica restituisce $true$ e nulla di piu'.

13/03/2017

3.4 Altri due predicati extralogici

Illustriamo l'utilizzo di due semplici predicati extralogici partendo da una semplice esempio

Esempio 1 : supponiamo di voler implementare un predicato nth che data una lista e una posizione restituisce l'elemento della lista in quella posizione. Se invece viene dato un elemento, restituisce la posizione di quell'elemento. Per implementarlo il codice e' il seguente

```
nth([Head|_], 0, Head):-!.
nth([_|Tail], Pos, X):-
   nonvar(Pos), !,
   Pos1 is Pos-1,
   nth(Tail, Pos1, X).
nth([_|Tail], Pos, X):-
   nth(Tail, Pos1, X),
   Pos is Pos1+1.
```

Nell'esempio appena visto abbiamo introdotto il predicato extralogico $nonvar(X)$. Tale predicato va a verificare se X non sia una variabile. Analogamente esiste il predicato extralogico $var(X)$ che va a verificare se X sia una variabile.
Nel caso dell'esempio l'idea e' la seguente: se e' la posizione a non essere una variabile iteriamo sulla posizione e tagliamo gli altri rami. Se invece Pos era una variabile allora richiamiamo un'altra regola nth che gestisce tale caso.

4 Strategia di ricerca in Prolog

Per andare a vedere come si possano sviluppare le strategie di ricerca in Prolog riassumiamo velocemente alcuni concetti chiave.
Un problema di ricerca e' definito da:

- Stato iniziale
- Insieme delle azioni: un'azione e' quell'evento che fa cambiare lo stato
- Insieme degli obiettivi
- Costo di ogni azione

Diremo inoltre che lo spazio degli stati e' costituito dallo stato iniziale e dall'insieme degli stati raggiungibili tramite le azioni. L'insieme degli stati che vengono attraversati andando da uno stato all'altro verra' detto *cammino* e ogni cammino avra' un suo *costo* dipendente dalle azioni effettuate.
Chiaramente una *soluzione* al problema sara' un cammino dallo stato iniziale ad uno stato obiettivo. Particolarmente interessante sara' la *soluzione ottima*, ovvero il cammino che ha il costo minimo.

Una domanda importante da porci adesso e': come rappresentare gli stati in Prolog? La risposta e' relativamente semplice: gli stati saranno rappresentati da termini. In generale saranno dei fatti o delle liste(se dovremo rappresentare informazioni composte).
Ogni azione verra' realizzata tramite due regole distinte: *applicabile* e *trasforma*. Vediamole piu' nel dettaglio:

- $applicabile(Az, S)$: verifica se l'azione Az e' applicabile allo stato S
- $trasforma(Az, S, NUOVO\_S)$: applica l'azione allo stato S ottenendo un nuovo stato chiamato $NUOVO_S$

Cerchiamo di ragionare sul problema di un labirinto. Ci vengono forniti il numero delle righe e il numero delle colonne. Ci viene fornita la posizione iniziale, la posizione degli ostacoli, e quello che sappiamo e' che possiamo spostarci in una casella adiacente tramite un'azione. Chiameremo nord est sud e ovest le azioni di spostamento con l'ovvio significato.
L'idea iniziale e' di controllare se gli spostamenti sono possibili, e nel caso applicarli. Potremmo allora esprimere anzitutto lo stato iniziale, quello finale e i vari ostacoli come segue:

```
% Esempio 10 x 10
num_col(10).
num_righe(10).
occupata(pos(2,5)).
occupata(pos(3,5)).
occupata(pos(4,5)).
occupata(pos(5,5)).
```

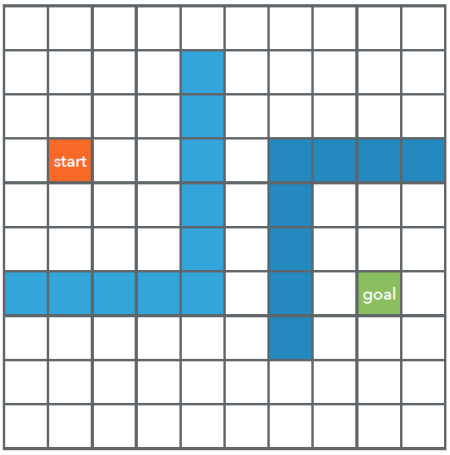

Figura 10: Struttura del labirinto

```
occupata(pos(6,5)).
occupata(pos(7,5)).
occupata(pos(7,1)).
occupata(pos(7,2)).
occupata(pos(7,3)).
occupata(pos(7,4)).
occupata(pos(5,7)).
occupata(pos(6,7)).
occupata(pos(7,7)).
occupata(pos(8,7)).
occupata(pos(4,7)).
occupata(pos(4,8)).
occupata(pos(4,9)).
occupata(pos(4,10)).
iniziale(pos(4,2)).
finale(pos(7,9)).
```

Mostriamo in Figura 10 la struttura del labirinto

A questo punto possiamo descrivere le nostre regole *applicabile* e *trasforma*. Diamo entrambe le regole ma solo per lo spostamento ad est. Gli altri spostamenti avranno regole del tutto simili.

```
applicabile(est,pos(R,C)) :-
   num_col(NC),
   C<NC,         //Controlliamo se siamo nell'ultima colonna
   C1 is C+1,
   \+ occupata(pos(R,C1)).
trasforma(est,pos(R,C),pos(R,C1)) :- C1 is C+1.
...
```

Quindi abbiamo fatto il ragionamento seguente: se non siamo nell'ultima colonna($C < NC$) e se la posizione ad est non e' occupata, allora possiamo dire che la regola est e' applicabile.
A questo punto, una volta che la regola e' applicabile, non dobbiamo fare ulteriori controlli, dunque la regola trasforma semplicemente crea un nuovo stato in cui ci siamo spostati di una colonna a destra.

14/03/2017

4.1 Ricerca in profondita'

Sviluppiamo adesso una strategia di ricerca vera e propria andando ad implementare una ricerca in profondita'

```
profondita(Soluzione):-
   iniziale(S),
   ric_prof(S, Soluzione).

ric_prof(S, []):-finale(S), !.
ric_prof(S, [Azione|AltreAzioni]):-
   applicabile(Azione, S),
   trasforma(Azione, S, SNuovo),
   ric_prof(SNuovo, AltreAzioni).
```

Abbiamo utilizzato anche un wrapper per invocare piu' comodamente la ricerca.
Notiamo che in pochissime righe di codice abbiamo implementato una ricerca in profondita'. Questo e' uno dei vantaggi dell'approccio ricorsivo in Prolog.
Andando ad invocare la query:
$? - profondita(ListaMosse)$.
possiamo verificare che questa soluzione va in loop.
Questo succede perche' il nostro robot, applicando sempre per prima la regola che lo sposta ad est, va a contatto con il muro alla sua destra. Non potendo applicare tale regola applica la seconda, che lo sposta ad ovest. Ma adesso si sposta di nuovo ad est visto che e' la prima regola applicabile...
Dunque dobbiamo introdurre un meccanismo che controlli se abbiamo gia' attraversato uno stato. Il codice che lo fa e' il seguente:

```
profonditaControlloCicli(Soluzione):-
   iniziale(S),
   ric_prof_CC(S, [S], Soluzione).

ric_prof_CC(S, _, []):- finale(S), !.
ric_prof_CC(S, Visitati, [Azione|AltreAzioni]):-
   applicabile(Azione, S),
   trasforma(Azione, S, SNuovo),
   \+member(SNuovo, Visitati),
   ric_prof_CC(SNuovo, [SNuovo | Visitati], AltreAzioni).
```

Andando ad invocare la query
$profonditaControlloCicli(ListaMosse)$.
Otteniamo effettivamente una lista di mosse da effettuare per raggiungere la soluzione della forma:

$[est, est, sud, ovest, ovest, ovest, nord, nord, est, est, est, nord, ovest, ovest, ovest, nord, est, est, est, est, est, est, est, est, est, sud, ovest, ovest, ovest, ovest, sud, sud, sud, sud, sud, sud, ovest, ovest, ovest, ovest, ovest, sud, est, est, est, est, est, est, est, est, est, nord, ovest, ovest, nord, est]$

Notiamo comunque che la soluzione non sembra essere proprio quella ottima, e che forse sarebbe una buona idea dare un limite alla profondita' della nostra ricerca. Proviamo allora ad implmentare un algoritmo di ricerca a profondita' limitata:

```
profonditaLimitata(MaxDepth, Soluzione):-
   iniziale(S),
   ric_prof_MaxDepth(S, [S], MaxDepth, Soluzione).

ric_prof_MaxDepth(S,_,_,[]):-finale(S),!.
ric_prof_MaxDepth(S, Visitati, MaxDepth, [Azione|AltreAzioni]):-
   MaxDepth > 0,
   applicabile(Azione, S),
   trasforma(Azione, S, SNuovo),
   \+member(SNuovo, Visitati),
   MaxDepth1 is MaxDepth-1,
   ric_prof_MaxDepth(SNuovo,[SNuovo|Visitati], MaxDepth1, AltreAzioni).
```

A questo punto invocando la query:
$profonditaLimitata(5, ListaMosse)$.
Otteniamo subito $false$. Questo e' normale visto che non esiste un cammino che raggiunga la soluzione in meno di cinque passi.
Se invece andassimo ad invocare la query:
$profonditaLimitata(20, ListaMosse)$
otterremmo il seguente risultato:

$[est, est, nord, nord, nord, est, est, sud, sud, sud, sud, sud, sud, sud, sud, est, est, est, nord, nord]$

E in effetti e' possibile verificare che e' una soluzione valida.

Prima di proseguire introduciamo due predicati extralogici che ci serviranno nel seguito, *assert* e *retract*:

- assert(x): aggiunge "al volo" il fatto x alla nostra base di conoscenza
- retract(x): rimuove "al volo" il fatto x alla nostra base di conoscenza

Prima di vedere un esempio che li impiega sottolineiamo che questi due predicati possono essere indispensabili talvolta ma che comunque devono essere usati con cautela.

Esempio 1 : supponiamo di aprire l'interprete Prolog e di eseguire la stringa:
$?- assert(livello(pippo))$.
true.
A questo punto abbiamo aggiunto il fatto $livello(pippo)$ alla nostra base di conoscenza. Andando ad eseguire la query
$?- livello(X)$.
$X = pippo$.
E questo e' del tutto normale visto che il fatto $livello(pippo)$ e' stato aggiunto alla base di conoscenza.

Supponiamo adesso di eseguire
$?- retractall(livello(\_))$.
true.
A questo punto abbiamo tolto tutti i fatti riguardanti i fatti *livello*(e dunque abbiamo rimosso il fatto $livello(pippo)$).
Se andassimo adesso a chiedere
$?- livello(X)$.
$false$. Questo perche' a questo punto non esistono piu' fatti relativi a *livello*

Dobbiamo anche dire una cosa importante riguardo al retract. È possibile effettuare il retract solo e soltanto di quei fatti che sono stati aggiunti "al volo" tramite una assert.
Dunque se avevamo una base di conoscenza originaria non possiamo effettuare retract su fatti della base di conoscenza.

15/03/2017

Vediamo un altro esempio che utilizzi la retract e la assert

Esempio 2 : supponiamo di voler implementare un'insieme di regole Prolog che calcolino la somma dei primi n elementi di una lista, con n noto. Il codice Prolog e':

```
sommaPrimiN(0, _, 0):-!.
sommaPrimiN(N, [Head|Tail], Somma):-
   N1 is N-1,
   sommaPrimiN(N1, Tail, SommaTail),
   Somma is SommaTail+Head.
```

Supponiamo invece di voler sommare i primi n elementi, con n **non** noto, affinche' la loro somma sia maggiore di un certo m che invece e' noto. Supponiamo che nel nostro caso tale m sia 30.

```
init:-assert(livello(1)).
somma(Lista, Somma):-
```

```
   livello(N),
   sommaPrimiN(N, Lista, Somma),
   Somma > 30,
   write('Somma maggiore di 30 con '),
   write(N),
   write(' elementi').
somma(Lista, Somma):-
   livello(N),
   retract(livello(_)),
   N1 is N+1,
   assert(livello(N1)),
   somma(Lista, Somma).
```

---

a questo punto invocando prima di tutto la regola
$? - init.$
$true.$
Andando ad effettuare la query:
$? - somma([8,9,3,4,3,1,5,6,7], Somma).$
$Sommamaggioredi30con7elementi$
$Somma = 33$
Notiamo che assumeremo di partire a contare almeno un elemento(grazie alla regola init che invoca la assert(livello(1)). Quello che abbiamo fatto e' dare due regole: nella prima siamo nel caso in cui effettivamente i primi N elementi della lista sommati danno un valore maggiore di 30. Nella seconda regola invece le cose non sono andate bene, e allora dobbiamo aumentare di 1 il livello. Dunque effettuiamo una retract che rimuove il livello corrente e aggiungiamo tramite la assert il livello aumentato di 1, andando ad invocare nuovamente la regola somma.

Torniamo nuovamente alle strategie di ricerca e cerchiamo di implementare una strategia di ricerca per risolvere il gioco dell'otto. Possiamo rappresentare lo stato di gioco tramite una lista che contenga in ordine dall'alto verso il basso e da sinistra verso destra i numeri. Nel codice che segue faremo uso della "funzione" nth che gia' avevamo definito in precedenza e della strategia di ricerca che avevamo gia' definito per la ricerca in profondita'.

```
%  -------
%  |2|4|3|
%  |7|1|6|
%  | |5|8|
%  -------
iniziale([2,4,3,7,1,6,vuoto,5,8]).
finale([1,2,3,4,5,6,7,8,vuoto]).

bordoSinistro(Posizione):-
   Resto is Posizione mod 3,
   Resto = 0.
```

```
bordoDestro(Posizione):-
   Resto is Posizione mod 3,
   Resto = 2.
bordoInferiore(Posizione):-
   Posizione > 5.
bordoSuperiore(Posizione):-
   Posizione < 3.
%sposto la tessera a sinistra dello spazio vuoto verso destra
applicabile(est,Stato):-
   nth(Stato, PosVuoto, vuoto),
   \+bordoSinistro(PosVuoto).
...
trasforma(sud, Stato, NuovoStato):-
   nth(Stato, PosVuoto, vuoto),
   PosTessera is PosVuoto-3,
   swap(Stato, PosVuoto, PosTessera, NuovoStato).
...
swap(Lista, Pos1, Pos2, NuovaLista):-
   nth(Lista, Pos1, X1),
   nth(Lista, Pos2, X2),
   setElementAt(Lista, Pos1, X2, Temp),
   setElementAt(Temp, Pos2, X1, NuovaLista).
setElementAt([_|Tail], 0, X, [X|Tail]):-!.
setElementAt([Head|Tail], Pos, Elem, [Head|NuovaTail]):-
   Pos1 is Pos-1,
   setElementAt(Tail, Pos1, Elem, NuovaTail).
```

Notiamo alcune cose: la mossa "est"(potremmo mostrare ragionamenti analoghi per tutte le altre) sposta la tessera che si trova alla sinistra dello spazio vuoto verso destra. Infatti notiamo che tale mossa puo' essere applicata a condizione che lo spazio vuoto non sia sul bordo sinistro del campo di gioco.
Notiamo inoltre che quando trasformiamo, ad esempio utilizzando la mossa sud, stiamo spostando la tessera al di sopra dello spazio vuoto verso il basso. In questo modo, data la rappresentazione tramite lista, risulta abbastanza evidente che le due posizioni da scambiare all'interno della lista saranno quella che contiene il valore "vuoto" e quella posizionata 3 posizioni prima(ovvero sulla riga superiore ma nella stessa colonna).
A questo punto eseguendo la query:
$? - profonditaLimitata(8, Soluzione).$
$Soluzione = [sud, ovest, sud, est, nord, ovest, nord, ovest]$
Che e' in effetti una soluzione per il problema dell'otto.

20/03/2017

4.2 Ricerca in ampiezza

Cerchiamo di implementare una nuova strategia di ricerca: la ricerca in ampiezza.
Implementarla e' un po' piu' complicato perche' non possiamo sfruttare appieno

la ricorsione, dobbiamo invece gestire per forza di cose una coda. Ci basiamo quindi su due idee:

1. Usare dei nodi strutturati diversamente:

   $$nodo(stato,\ lista\ azioni\ prcedenti)$$

   Per capirlo meglio vediamo un brevissimo esempio

   Esempio 1 : nel caso del gioco dell'otto un tipico nodo potrebbe essere:
   $nodo([8,2,3,vuoto,1,4,7,6,5],\ [est,\ nord,\ ovest])$

2. Creare una lista di nodi: in questo modo potremo riprendere i vari rami che abbiamo abbandonato per esplorare in ampiezza

Prima di passare all'implementazione vera e propria introduciamo un nuovo predicato chiamato *findall*.
La sintassi del findall e':

$$findall(azione,\ goal,\ ListaAzioni)$$

La semantica del findall e' che inseriamo nella $ListaAzioni$ tutte le possibili istanziazioni di *azione* tali che soddisfano *goal*.

Esempio 2 : supponiamo di voler vedere quali siano le azioni eseguibili in un certo momento del gioco dell'otto. Possiamo eseguire il findall
$?-findall(Azione, applicabile(Azione, [2,3,5,1,7,8,4,6,vuoto]), ListaAzioni)$
$ListaAzioni = [est, sud]$
Quello che abbiamo fatto e' chiederci quali siano le azioni applicabili allo stato corrente(ovvero quello in cui lo spazio vuoto e' nella casella in basso a destra). In tal caso la risposta e' composta dalle azioni *est* e *sud*, ovvero le uniche due mosse che soddisfano la condizione di applicabilita'.

Vediamo adesso l'implementazione della ricerca in ampiezza:

```
ampiezza(Soluzione):-
   iniziale(S),
   ric_ampiezza([nodo(S, [])], Soluzione).

%ric ampiezza(coda nodi da espandere, Lista azioni Soluzione)
ric_ampiezza([nodo(S, ListaAzioni)|_], ListaAzioni):-
   finale(S),!.
ric_ampiezza([nodo(S, ListaAzioni)| RestoDellaCoda], Soluzione):-
   espandi(nodo(S, ListaAzioni), ListaSuccessoriS),
   append(RestoDellaCoda, ListaSuccessoriS, NuovaCoda),
   ric_ampiezza(NuovaCoda, Soluzione).

espandi(nodo(S, ListaAzioni), ListaSuccessoriS):-
   findall(Az, applicabile(Az, S), ListaApplicabili),
   successori(nodo(S, ListaAzioni), ListaApplicabili, ListaSuccessoriS).
```

```
successori(_,[],[]):-!.
successori(nodo(S, ListaAzioni), [Az|AltreAzioniApplicabili],
    [nodo(SNuovo, ListaAzioniSNuovo)| AltriSuccessoriS]):-
   trasforma(Az, S, SNuovo),
   append(ListaAzioni, [Az], ListaAzioniSNuovo),
   successori(nodo(S, ListaAzioni), AltreAzioniApplicabili,
       AltriSuccessoriS).
```

In questo caso l'idea e' quella di avere una coda di nodi da esplorare, e ogni volta che analizziamo un nuovo nodo n, aggiungiamo in coda alla lista tutti quei nodi raggiungibili dal nodo n mediante un'azione.
Questa soluzione, come gia' quella della ricerca in profondita', e' di carattere generale e puo' essere applicata a tutti quei problemi basati sulle regole *applicabile* e $trasforma$.

4.3 Strategia di ricerca informate

Fino ad ora abbiamo visto una serie di strategie non informate come la ricerca in profondita', la ricerca in profondita' limitata, l'iterative deepening e la ricerca in ampiezza.
Esistono pero' anche delle strategie di ricerca informate, ovvero che utilizzano una *funzione euristica* $h(n)$. Di fatto $h(n)$ e' una funzione che dovrebbe stimare il costo per raggiungere la soluzione dallo stato attuale.
Andando a definire come $g(n)$ il costo di un cammino dallo stato iniziale al nodo n allora possiamo utilizzare un algoritmo A^* basato sulla seguente funzione:

$$f(n) = g(n) + h(n)$$

L'idea di questa funzione da minimizzare e' che cerchiamo di muoverci verso quello stato che ci permette di raggiungere la soluzione a costo minore. A questo costo contribuiscono sia il costo per raggiungere quello stato(ovvero $g(n)$) sia il costo per raggiungere la soluzione da tale stato(ovvero $h(n)$).
Similmente opera l'algoritmo IDA^* che fondamentalmente e' una ricerca in profondita' che sfrutta un'euristica.
Ad ogni modo per ulteriori dettagli e' possibile consultare il libro di testo *Artificial Intelligence, a modern approach* di Stuart J. Russell, Peter Norvig.

21/03/2017

5 Answer Set Programming

L'*answer set programming*(ASP) e' una metodologia di programmazione nata nel momento storico in cui si cercava di trovare una semantica per la negazione per fallimento. In seguito e' diventata qualcosa di piu', ovvero il fondamento di una nuova tecnica di programmazione.
Cominciamo dicendo che nell'ASP quello che cambia e' l'approccio: nel Prolog

la soluzione e' la prova e viene trovata andando a cercare regole che unifichino. Questo non succede nell'ASP in cui le soluzioni sono qualcosa di simile ai *modelli* della logica classica e non li andiamo a cercare con un procedimento di unificazione guidato dal goal.
Questo approccio molto diverso e' utile per andare a risolvere alcuni problemi combinatori come il soddisfacimento di vincoli. Questo perche' nel Prolog asseriamo quello che sappiamo, mentre nell'ASP do dei veri e propri vincoli. Ad ogni modo gli elementi basilari dell'ASP sono:

- Regole nella forma Prolog $a :- b_1, \ldots, b_n$
- Letterali affermati o negati. La negazione in ASP puo' essere di due tipi
  1. -p: indica la negazione classica
  2. not p: indica la negazione per fallimento

Una particolarita' e' che nelle regole non e' obbligatorio che vi sia la testa. Quando questo non succede allora ho un *integrity constraint*. Con questo intendiamo dire che
$:- a_1, \ldots, a_n$
e' inconsistente solo se $a_1, \ldots, a_n$ sono tutti veri.
Un altro fatto importante e' che l'ASP si puo' applicare soltanto ai programmi logici proposizionali. In ASP possiamo usare delle variabili ma le clausole devono poter essere trasformate in un numero finto di clausole ground.
Vediamo invece le piu' importanti differenze tra Prolog e ASP:

- in ASP l'ordine dei letterali non conta. Questo perche' non esiste un interprete che debba scorrere le regole come succedeva nel Prolog
- Prolog e' goal-directed, ASP no. ASP cerca invece un modello
- ASP non essendo goal-directed non puo' andare in loop. Questo perche' cerca solo delle istanziazioni possibili per le variabili
- Prolog possiede il cut, mentre ASP no.

Poniamo di nuovo l'attenzione sugli operatori di negazione - e *not*. Notiamo che sono due operatori molto diversi tra loro.
La negazione classica e' di fatto piu' forte. Supponendo di avere $-p$ sapremo che questa clausola e' vera solo se sappiamo esplicitamente che $p = false$.
Nella negazione per fallimento invece restituiremmo $true$ anche nel caso in cui non si possiedano informazioni su p. Vediamolo meglio su qualche esempio

Esempio 1 : consideriamo il seguente programma ASP

```
fly(X):-bird(X), not abnormal(X).
bird(X):-penguin(X).
abnormal(X):- penguin(X).
penguin(tux).
bird(tweety).
```

Andando ad eseguire questo programma all'intero di CLINGO, la risposta che otteniamo e'

Answer : 1
$bird(tweety)$ $bird(tux)$ $abnormal(tux)$ $fly(tweety)$ $penguin(tux)$
$SATISFIABLE$

Quello che ci sta dicendo CLINGO e' che abbiamo trovato un possibile modello che e' soddisfacibile, e che le informazioni ricavabili sono $bird(tweety), \ldots$
Andando a scrivere il seguente programma

```
fly(X):-bird(X), not abnormal(X).
bird(X):-penguin(X).
abnormal(X):-penguin(X).
penguin(tux).
bird(tweety).
penguin(tweety).
```

quello che otteniamo e'

Answer : 1
$bird(tweety)$ $bird(tux)$ $abnormal(tux)$ $abnormal(tweety)$
$penguin(tux)$ $penguin(tweety)$
$SATISFIABLE$

È successo quello che ci potremmo aspettare. Aggiungendo l'informazione che *tweety* e' un pinguino, scopriamo le stesse cose di prima eccetto una: $fly(tweety)$.

Esempio 2 : consideriamo sempre il campo dei volatili ma con il seguente programma

```
fly(X):-bird(X), not -fly(X).
bird(X):-penguin(X).
-fly(X):-penguin(X).
bird(tweety).
penguin(tux).
```

In questo caso la risposta e'

Answer : 1
$bird(tweety)$ $penguin(tux)$ $bird(tux)$ $-fly(tux)$ $fly(tweety)$
$SATISFIABLE$

Usando la negazione per fallimento seguita dalla negazione classica otteniamo un risultato(ovviamente) diverso. La particolarita' in questo caso

e' che non solo sappiamo che *tweety* vola, ma come risulta dalla risposta sappiamo anche che *tux* non vola!

5.1 Semantica dell'ASP

A questo punto possiamo parlare un po' della semantica nell'ASP.
Un answer set di fatto e' un insieme minimale di istanze che verificano un certo insieme di regole.

Esempio 1 : supponiamo di avere queste due regole ASP
$p :- q.$
$q.$
In questo caso l'answer set e' $\{p, q\}$. Dunque vogliamo dire che affinche' le regole scritte sopra siano vere devono essere vere sia p che q.
Nel caso in cui le regole siano
$p :- q, r.$
$q.$
allora l'answer set e' $\{q\}$. Notiamo gia' ora che se non abbiamo informazioni su una certa variabile la consideriamo istanziata a false. Per questo motivo q, r e' false e l'intera implicazione e' verificata.

Dobbiamo allora cercare un algoritmo che determini l'answer set di un programma ASP.
Per farlo introduciamo il *ridotto* P^S di un programma P rispetto ad un insieme di atomi S. Dato l'insieme degli atomi S, il ridotto P^S viene determinato applicando due semplici regole

- Rimuovendo per intero ogni regola il cui corpo contiene $notL$ con $L \in S$
- Rimuovendo tutti i restanti $notX$ con X qualunque dalle restanti regole

A questo punto per costruzione P^S non contiene atomi con negazione per fallimento.
L'idea dell'algoritmo per determinare l'answer set e' che una volta stabilito S, e calcolato P^S, se S e P^S coincidono allora S e' un answer set per il programma P. Vediamo un esempio per capire meglio cosa questo voglia dire

Esempio 2 : supponiamo di avere il seguente programma ASP e di volerne determinare un answer set

```
p:-a.
a:-not b.
b:-not a.
```

I possibili insiemi S sono tutti i sottoinsiemi di $\{a, b, p\}$. Studiamo alcuni dei casi possibili:

1. Insieme $\{\emptyset\}$: il ridotto di $P^\emptyset$ e'
   $p :- a.$

a.
b.
Infatti applicando la prima regola non eliminiamo nessuna regola per intero(poiche' nessun letterale appartiene ad S), pero' eliminiamo tutti i not e dunque eliminiamo entrambi i corpi della seconda e della terza regola. A questo punto determiniamo un answer set per $P^{\emptyset}$:
$P^{\emptyset} = \{a, b, p\}$
Questo perche' i fatti devono essere necessariamente veri ma se a e' vero, allora deve esserlo anche p perche' senno' la prima regola sarebbe falsa.
Poiche' $\emptyset \neq \{a, b, p\}$ allora $\emptyset$ non e' un answer set per il nostro programma P

2. Insieme $\{a\}$: il suo ridotto e'
   $p :- a.$
   $a.$
   Un answer set per P^S e' $\{a, p\}$. Poiche' P^S e' diverso da S anche $\{a\}$ non e' un answer set per P
3. Insieme $\{b\}$: in questo caso il ridotto e'
   $p :- a.$
   $b.$
   Answer set per $P^S = \{b\}$. In questo caso l'answer set per P^S coincide con S dunque $\{b\}$ e' effettivamente un answer set per P
4. Insieme $\{p\}$: in questo caso il ridotto e': $p :- a.$
   $a.$
   $b.$
   In questo caso l'answer set per P^S e' $\{a, b, p\}$. Essendo diverso da S allora S non e' answer set per P
5. Insieme $\{a, b\}$: in questo caso il ridotto e'
   $p :- a.$
   Un answer set per P^S e' $\{\emptyset\}$. Anche in questo caso S non e' answer set per P

Potremmo procedere oltre andando a vedere i restanti casi. Ci accontentiamo di dire che sono analoghi e che in conclusione gli unici answer set per P sono $\{b\}$ e $\{a, p\}$
Questo fatto viene mostrato anche da CLINGO

Answer : 1
a p
Answer : 2
b
SATISFIABLE

22/03/2017

Vediamo qualche altro esempio per comprendere meglio di cosa stiamo parlando

Esempio 3 : consideriamo il problema del Nixon Diamond. Il noto presidente americano Nixon e' un quacchero ma e' anche un repubblicano. È noto che i quaccheri sono pacifisti e che i repubblicani non lo sono. Allora Nixon e' o non e' un pacifista? Formalizziamo il problema

```
pacifist(X):- quaker(X), not -pacifist(X).
-pacifist(X) :- repubblican(X), not pacifist(X).
repubblican(nixon).
quaker(nixon).
```

La prima regola ammettiamo che se una persona e' quacchero e non riusciamo a reperire informazioni sul fatto che non sia pacifista, allora diremo che quella persona e' pacifista. La seconda regola invece dice che per un repubblicano, a meno che non si riesca a reperire esplicitamente il fatto che e' pacifista, daremo per scontato che non sia un pacifista. La soluzione che troviamo con CLINGO e' la seguente

$Answer : 1$
$repubblican(nixon)\ quaker(nixon)\ pacifist(nixon)$
$Answer : 2$
$repubblican(nixon)\ quaker(nixon)\ -pacifist(nixon)$
$SATISFIABLE$

Ovvero abbiamo ottenuto che Nixon e' in due modelli diversi sia pacifista che non pacifista. In questo caso la soluzione e' dunque poco significativa. Infatti in generale, per essere vero qualcosa deve esserlo in tutti i modelli.

Consideriamo adesso una cosa. In realta' potremmo non usare esplicitamente la negazione classica ma sostituirla con qualcosa del tipo

```
pacifist(X):- quaker(X), not nonpacifist(X).
nonpacifist(X) :- repubblican(X), not pacifist(X).
repubblican(nixon).
quaker(nixon).

:-pacifist(X), nonpacifist(X).
```

Ovvero abbiamo definito un nuovo predicato nonpacifist. Il predicato nonpacifist e' pero in relazione con il predicato pacifist(uno e' l'opposto dell'altro). Quindi questa situazione deve essere segnalata mediante l'integrity constraint : $-pacifist(X), nonpacifist(X)$.

Nel seguito useremo spesso anche gli *aggregati*. Essendo molto piu' facili da implementare che da spiegare vediamo subito un esempio

Esempio 4 : un esempio di aggregato e'
$1\{a; b; c\}2$.

Con questo indichiamo che per il nostro modello dobbiamo prendere almeno 1 degli argomenti in parentesi ma non piu' di 2. Provando ad eseguire quest'unica riga in CLINGO otteniamo sei modelli diversi: a, b, c, $[a, b]$, $[b, c]$, $[c, a]$.
Scrivendo invece $\{a; b; c\}2$. troviamo sette modelli(i sei di prima piu' quello vuoto).
Scrivendo invece $2\{a; b; c\}$. troviamo i quattro modelli $[a, b]$, $[b, c]$, $[c, a]$, $[a, b, c]$.

Vediamo un uso un pochino piu' furbo degli aggregati con un nuovo tipo di aggregato

Esempio 5 : supponiamo di aver scritto il seguente codice

```
1{p(X): q(X)}1.
q(1).
q(2).
q(3).
q(mario).
q(gianni).
```

Questo nuovo tipo di aggregato $1\{p(X) : q(X)\}1$. indica che il predicato $p(X)$ puo' prendere fino a 1 solo valore che assume il predicato $q(X)$. Andando ad eseguire il codice infatti CLINGO trova 5 modelli diversi in cui $p(X)$ viene verificato con tutti i valori associati al predicato q. Ovvero 1,2,3,mario e gianni.

Prima di vedere un paio di esempi che riassumano quanto detto finora diciamo che possiamo specificare a CLINGO una direttiva che ci permetta di stampare solo una parte dei risultati ottenuti. Questo ci permette di evitare di stampare ogni volta cose che gia' sapevamo, come ad esempio tutti i fatti che abbiamo scritto. La notazione e'

$$p/n$$

in questo modo stamperemo solo il predicato p ad n argomenti[6].
Vediamo allora qualche esempio riassuntivo

Esempio 6 : supponiamo di essere su un'isola composta di due soli tipi di persone

- Onesti: dicono sempre la verita'
- Bugiardi: dicono sempre e solo bugie

Incontriamo due persone a e b. La persona a ci rivela che sia a che b sono bugiardi. Come classificarli allora? Scriviamo un programma ASP

[6]Notiamo che e' necessario specificare il numero degli argomenti del predicato perche' in CLINGO come in Prolog predicati aventi stesso nome ma numero di argomenti diverso sono considerati predicati diversi

```
persona(a).
persona(b).
tipo(onesto).
tipo(bugiardo).

%ogni persona ha un solo tipo
1 {ha_tipo(P,T): tipo(T)} 1 :- persona(P).

%asserzione di a
a_dice_il_vero :- ha_tipo(a, bugiardo), ha_tipo(b, bugiardo).

%vincoli
:-ha_tipo(a, bugiardo), a_dice_il_vero.
:-ha_tipo(a, onesto), not a_dice_il_vero.

#show ha_tipo/2.
```

L'aggregato ci permette di specificare che le persone possono essere solo oneste o bugiarde. Notiamo che anche in questo caso i vincoli sono rappresentati da degli integrity costraints. In particolare il primo vincolo indica che a non puo' essere allo stesso tempo bugiardo e dire il vero. Il secondo vincolo indica invece che a non puo' essere onesto e non dire il vero. Notiamo che abbiamo usato # show ha_tipo/2 per stampare solo i risultati che ci interessano. Infatti andando ad eseguire il programma in CLINGO otteniamo che

$Answer : 1$
$ha\_tipo(a, bugiardo) ha\_tipo(b, onesto)$

Ovvero abbiamo ottenuto che a e' un bugiardo mentre b e' onesto.

Consideriamo un'altra formulazione dello stesso problema in cui abbiamo davanti tre persone a, b, c. La persona a dice che b e c sono onesti. La persona b dice che a e' bugiardo e c e' onesto. Come classificare in questo caso? Scriviamo un nuovo programma ASP

```
persona(a; b; c).
tipo(onesto).
tipo(bugiardo).

%ogni persona ha un solo tipo
1 {ha_tipo(P,T): tipo(T)} 1 :- persona(P).

%asserzioni di a e di b
dice_il_vero(a) :- ha_tipo(b, onesto), ha_tipo(c, onesto).
dice_il_vero(b) :- ha_tipo(a, bugiardo), ha_tipo(c, onesto).
```

```
%vincoli
:-ha_tipo(P, bugiardo), dice_il_vero(P).
:-ha_tipo(P, onesto), not dice_il_vero(P).

#show ha_tipo/2.
```

Notiamo che abbiamo usato la notazione persona(a; b; c) che ci permette di specificare che a, b e c sono tutte e tre persone. Abbiamo inoltre generalizzato l'integrity constraint per poter imporre i vincoli su tutte e tre le persone in un colpo solo. In questo caso il risultato che otteniamo e'

Answer : 1
$ha\_tipo(a, bugiardo)$ $ha\_tipo(b, bugiardo)$ $ha\_tipo(c, bugiardo)$

Abbiamo dunque ottenuto che tutte e tre le persone in questo caso erano bugiarde.

22/03/2017

Procediamo con qualche altro esempio su CLINGO

Esempio 7 : supponiamo di voler risolver un problema di cripto aritmetica. Con questo intendiamo di avere due parole che sommate danno una nuova parola. Lo scopo e' trovare un assegnamento possibile per ogni lettera affinche' la "somma" sia corretta. Useremo le parole *send, more* che sommate dovranno dare *money*.

```
% SEND+
% MORE=
%-----
%MONEY
cifra(0..9).
lettera(s; e; n; d; m; o; r; y;).

%assegno ad ogni lettera una cifra
1{assegna(L,C) : cifra(C)} 1 :- lettera(L).

%assegno ad ogni cifra al piu una lettera
{assegna(L,C) : lettera(L) } 1:- cifra(C).

%GOAL
somma :- cifra(S; E; N; D; M; O; R; Y;),
   assegna(s, S), assegna(e, E), assegna(n,N),
   assegna(d, D), assegna(m, M), assegna(o, O),
   assegna(r, R), assegna(y, Y),
   (S*1000+E*100+N*10+D)+(M*1000+O*100+R*10+E) ==
       M*10000+O*1000+N*100+E*10+Y,
   S>0, M>0.

:- not somma.
```

Quello che abbiamo fatto e che spesso ci ritroveremo a fare e': specificare le regole e i fatti basilari(cifra, lettera...), specificare degli aggregati e dare un goal. L'ultima clausola serve per "avviare" la computazione. Infatti e' vera soltanto se una delle clausole del corpo e' falsa. E l'unica clausola e' proprio somma che infatti il programma va a calcolare.
Per andare un po' piu' nello specifico, in questo programma abbiamo specificato che ad ogni lettera deve essere associata una e una sola cifra, che ogni cifra non puo' essere associata a piu' lettere, e che dobbiamo comunque assegnare ad ogni lettera una certa cifra. A questo punto basta imporre tramite la notazione posizionale che valga l'equazione $(S*1000+E*100+N*10+D)+(M*1000+O*100+R*10+E) == M*10000+O*1000+N*100+E*10+Y$.
Le ultime due condizioni $S>0, M>0$ sono fondamentali perche' stabiliscono che le cifre piu' significative di ogni riga debbano essere maggiori di 0.
Andandolo ad eseguire otteniamo
Answer : 1
$assegna(o,0)$ $assegna(m,1)$ $assegna(s,9)$ $assegna(r,8)$ $assegna(n,6)$
$assegna(e,5)$ $assegna(d,7)$ $assegna(y,2)$
È possibile verificare che in effetti, con tali assegnamenti la somma e' corretta.

Esempio 8 : supponiamo di avere il problema del cambio della ruota. Dobbiamo sostituire una ruota forata rimuovendola dall'asse e inserendo una ruota di scorta. Anche qui abbiamo un problema di soddisfacimento di vincoli in quanto dobbiamo eseguire le azioni in un certo ordine(non possiamo mettere la ruota di scorta prima di aver tolto quella forata ad esempio). Il codice del programma ASP e'

```
#const lastlev=1.
livello(0..lastlev).

%ad ogni livello eseguo almeno una azione tra quelle disponibili
1{rimuovi(scorta, bagagliaio,S); monta(scorta, asse, S);
    rimuovi(bucata, asse, S)}:-livello(S).

%EFFETTI DELLE AZIONI
posizione(scorta, asse, S+1):-
   monta(scorta, asse, S),
   livello(S).
-posizione(scorta, pavimento, S+1):-
   monta(scorta, asse, S),
   livello(S).
posizione(scorta, pavimento, S+1):-
   rimuovi(scorta, bagagliaio, S),
   livello(S).
-posizione(scorta, bagagliaio, S+1):-
   rimuovi(scorta, bagagliaio, S),
```

```
   livello(S).
-posizione(bucata, asse, S+1):-
   rimuovi(bucata, asse, S),
   livello(S).
posizione(bucata, pavimento, S+1):-
   rimuovi(bucata, asse, S),
   livello(S).

%VINCOLI
:- rimuovi(scorta, bagagliaio, S), not posizione(scorta,
    bagagliaio, S).
:- rimuovi(bucata, asse, S), not posizione(bucata, asse, S).
:- monta(scorta, asse, S), not posizione(scorta, pavimento, S).
:- monta(scorta, asse, S), not posizione(bucata, asse, S).

%PERSISTENZA
posizione(Oggetto, Pos, S+1):-
   posizione(Oggetto, Pos, S),
   livello(S),
   not -posizione(Oggetto, Pos, S+1).
-posizione(Oggetto, Pos, S+1):-
   -posizione(Oggetto, Pos, S),
   livello(S),
   not posizione(Oggetto, Pos, S+1).

%STATO INIZIALE
posizione(bucata, asse, 0).
posizione(scorta, bagagliaio, 0).
-posizione(bucata, pavimento, 0).
-posizione(scorta, pavimento, 0).
-posizione(scorta, asse, 0).

%GOAL!
goal :- posizione(scorta, asse, lastlev+1).
:- not goal.

#show rimuovi/3.
#show monta/3.
#show posizione/3.
#show -posizione/3.
```

Notiamo che tramite la direttiva #const abbiamo stabilito quanti livelli al massimo possiamo attraversare. Ad ogni livello andiamo ad eseguire una o piu' azioni. Per essere piu' precisi, tramite l'aggregato abbiamo stabilito che ad ogni livello dobbiamo eseguire almeno una azione, ma volendo anche piu' di una in parallelo.
Abbiamo anche creato delle regole di persistenza che permettono di conservare dei fatti tra un livello e il successivo. COnsideriamo la prima regola di persistenza. Quello che dice e' che se un certo oggetto era in una certa

posizione al livello S e non riceviamo esplicitamente delle notizie di spostamento, allora al livello $S+1$ l'oggetto sara' ancora nella stessa posizione. Notiamo inoltre che ogni azione ha due effetti. Quindi inseriamo due regole. Consideriamo ad esempio la prima azione. Quando montiamo la ruota di scorta sull'asse, la posizione della ruota di scorta diventa l'asse(regola 1) e non e' piu' vero che la ruota di scorta sta sul pavimento. Ovviamente questi ragionamenti si applicano anche alle regole di persistenza e agli stati iniziali.
Notiamo infine che i vincoli rappresentano tutte le situazioni in cui non vogliamo trovarci. Ad esempio il primo vincolo indica che non possiamo rimuovere la ruota di scorta dal bagagliaio se la ruota non e' nel bagagliaio.
Andando ad eseguire il programma otteniamo il seguente risultato
$Answer : 1$
$posizione(bucata, asse, 0)$ $posizione(scorta, bagagliaio, 0)$
$-posizione(bucata, pavimento, 0)$ $-posizione(scorta, pavimento, 0)$
$-posizione(scorta, asse, 0)$ $posizione(scorta, asse, 2)$ $monta(scorta, asse, 1)$
$-posizione(scorta, pavimento, 2)$ $posizione(scorta, pavimento, 1)$
$rimuovi(scorta, bagagliaio, 0)$ $-posizione(scorta, bagagliaio, 1)$
$-posizione(scorta, bagagliaio, 2)$ $posizione(bucata, asse, 1)$
$-posizione(bucata, pavimento, 2)$ $posizione(bucata, asse, 2)$
$-posizione(bucata, pavimento, 1)$ $-posizione(scorta, asse, 1)$

03/04/2017

6 Introduzione agli agenti intelligenti

Andiamo a studiare sia il termine *agente* che il termine *intelligente*.
Cosa e' un'agente? Di questo termine si tende ad abusare ma possiamo vedere l'agente da due punti di vista diversi.
L'agente puo' essere visto in modo astratto come la concettualizzazione di un sistema complesso.
Un altro punto di vista dell'agente e' quello pratico in cui vediamo l'agente come un'entita' in grado di prendere delle decisioni non banali. Le decisioni non banali sono tutte quelle decisioni che non sono facilmente esprimibili tramite algoritmi perche' dipendenti dall'ambiente in cui e' collocato l'agente. Infatti l'agente percepisce l'ambiente e tramite gli attuatori lo modifica.
Sottolineiamo la differenza tra agente e algoritmo: un algoritmo segue il semplice schema:
$input \rightarrow Algoritmo \rightarrow output$
La struttura di un agente e' invece piu' complessa. Dipende dalle percezioni e dal modello dell'ambiente in possesso dell'agente. La struttura di un agente viene mostrata in Figura 11

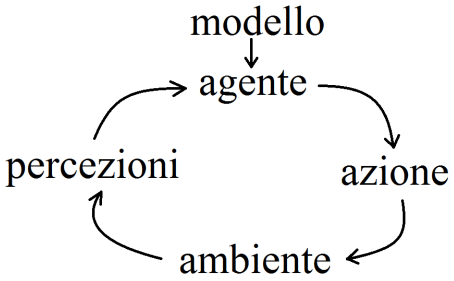

Figura 11: Struttura di un agente

Concentriamoci adesso sul termine *intelligente*.
Diremo che un'agente e' intelligente quando e' razionale, ovvero quando in presenza di una percezione e di un modello l'agente prende la decisione *giusta*. La decisione giusta e' quella che massimizza la *misura di prestazione*. La misura di prestazione deve essere definita in base alle proprieta' dell'ambiente e deve essere oggettiva. Perche' e' su di essa che andiamo a definire la *funzione obiettivo* che dovremo massimizzare.

Esempio 1 : supponiamo di avere un robot collocato in una stanza contenente dello sporco. Il robot puo' spostarsi e pulire la stanza. La misura di prestazione potrebbe essere il numero di celle pulite

Adesso che abbiamo illustrato i termini di agente e intelligente possiamo vedere i vari tipi di agenti intelligenti esistenti

- Agente reattivo semplice: possiede un insieme di regole *condizione-azione* e si basa solo sulla percezione corrente. In questo caso il confine tra agente e algoritmo e' abbastanza labile. Funziona bene solo in ambienti totalmente osservabili e statici

- Agente reattivo basato su modello: permette di affrontare ambienti parzialmente osservabili e dinamici perche' mantiene un suo stato interno che viene aggiornato col passare del tempo grazie alle osservazioni e grazie al modello del mondo. Quello che deve sapere e' come cambia il mondo per effetto delle azioni

- Agente basato su obiettivi: un agente ad obiettivi e' gia' completamente differente da un agente reattivo semplice. Infatti nell'agente reattivo semplice il fronte temporale ha estensione 1(considero solo il fatto corrente). Nell'agente con obiettivi invece il fronte temporale deve estendersi fino al raggiungimento dell'obiettivo. Quindi l'agente deve tenere traccia di possibili percorsi alternativi.

- Agente basato su utilita': utilizza un *piano* ovvero uno schema di azioni che permetta di massimizzare/minimizzare uno o piu' parametri(tempo, costo...)

Un'ultima ma non meno importante caratteristica e' che l'agente deve essere autonomo. L'agente e' autonomo se e' in grado di sostenere un *processo decisionale*. Tale processo consta di due parti

1. Information gathering: recupero delle informazioni tramite le percezioni
2. Apprendimento: capacita' di modificare il proprio modello del mondo

7 Sistemi esperti, CLIPS

7.1 Teoria dei sistemi esperti

I *sistemi esperti* o *sistemi a regole di produzione* sono sistemi software che hanno avuto un periodo d'oro verso la fine degli anni '70 inizio anni '80. I due sistemi esperti piu' famosi sono *Mycin* utilizzato per rilevare le infezioni del sangue e *R1/Xcon* utilizzato per supportare la costruzione di architetture hardware di grandi dimensioni in base alle preferenze del cliente.
Lo schema generale di un sistema esperto e' cosi' strutturato: un utente fornisce dei fatti al sistema che a sua volta fornisce delle *expertise* ovvero delle risposte per l'utente. I vantaggi dei sistemi esperti sono molteplici

- Sempre disponibile
- Puo' trarre conoscenza da campi diversi per fornire le proprie risposte
- Puo' dare una spiegazione delle decisioni che ha preso mostrando le regole che ha utilizzato per decidere

Dunque da cosa e' costituito un sistema esperto? Essenzialmente consta di tre parti

1. Base di conoscenza KB: l'insieme delle regole che il sistema dovra' applicare
2. Working Memory WM: contiene le informazioni che vengono utilizzati per stabilire quale regola applicare. I fatti immessi dall'utente vengono inseriti nella working memory
3. Motore inferenziale IE: componente che comunica con la KB e con la WM per stabilire quali azioni eseguire

Lo schema di un sistema esperto e' mostrato in Figura 12.

Andiamo a studiare meglio le varie parti del nostro sistema esperto.
La KB contiene un'insieme di regole della forma
$if(cond)then\ azione$
la condizione puo' essere una congiunzione di fatti. Quando un insieme di fatti contenuti nella WM permettono di verificare la *cond* di una certa regola, allora tale regola e' detta *attivabile*. Notiamo che nei sistemi esperti vale l'assunzione di mondo chiuso, ovvero che se un fatto non e' contenuto all'interno della nostra

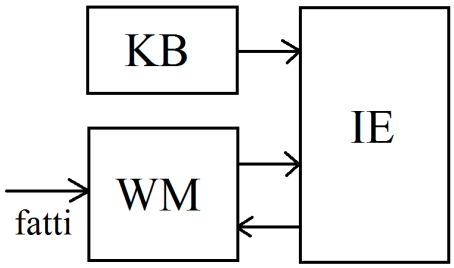

Figura 12: Struttura di un sistema esperto

WM allora consideriamo quel fatto come se fosse falso.
Le regole contenute all'interno della KB sono *esperienze autocontenute*, ovvero indipendenti fra loro

Esempio 1 : consideriamo le due seguenti regole

$if(piove)thenprendiOmbrello()$
$if(piove)thenmettiStivali()$

Queste due regole hanno la stessa condizione di attivazione ma l'effetto dell'una e' indipendente dall'effetto dell'altra

Le azioni che possono essere eseguite all'interno di un sistema esperto sono le seguenti con l'ovvio significato:
$ADD(fatto)$
$DEL(fatto)$
$MODIFY(fatto, modifica)$

Consideriamo adesso il motore inferenziale. Questa componente esegue un loop in tre fasi

1. Pattern matching: verifichiamo quali siano le regole della KB che verificano la propria condizione di attivazione utilizzando i dati della WM. Le regole attivabili vengono poste in un'*agenda* di cui parleremo fra poco.

2. Conflict resolution: stabiliamo quale regola eseguire fra tutte quelle attivabili

3. Execution: eseguiamo la regola

Mostriamo lo schema seguito dal motore inferenziale in Figura 13

L'agenda in cui vengono messe le regole attivabili puo' in certo momento contenere un numero di regole

- Nessuna regola: in questo caso il motore si ferma. Generalmente e' un brutto segno perche' tendenzialmente implica un fallimento

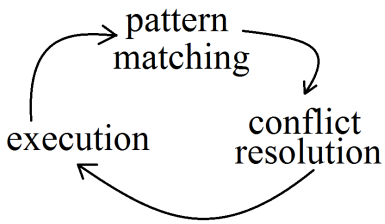

Figura 13: Schema del loop di esecuzione del motore inferenziale

- Una regola: la fase di conflict resolution sara' istantanea visto che sceglieremo l'unica regola disponibile
- Molte regole: nella fase di conflict resolution dovremo stabilire quale regola eseguire effettivamente

Esempio 2 : supponiamo di avere a disposizione la seguente KB

$R1: \ if(A) then ADD(D)$
$R2: \ if(A \wedge B) then DEL(A)$
$R3: \ if(A \wedge C) then DEL(C)$

Supponiamo di avere inizialmente la seguente WM: $WM_0 = \{A, B, C\}$.

Andando ad applicare le regole in ordine abbiamo che

$$WM_0 = \{A, B, C\} \underset{R1}{\longrightarrow} WM_1\{A, B, C, D\}$$

Notiamo che dopo questa prima esecuzione, se andiamo a verificare quali siano le nuove regole applicabili entreremo in un loop perche' la regola R1 e' nuovamente applicabile e lo sare' in tutte le future iterazioni

Per evitare le situazioni di loop viste nell'esempio precedente i sistemi esperti usano la tecnica di *rifrazione* che impedisce di attivare una stessa regola più volte usando gli stessi fatti

Esempio 3 : consideriamo nuovamente l'esempio di prima. Utilizzando la rifrazione le memorie vengono modificate come segue

$$WM_0 = \{A, B, C\} \underset{R1}{\longrightarrow} WM_1\{A, B, C, D\} \underset{R2}{\longrightarrow} WM_2\{B, C, D\}$$

Notiamo che dopo l'applicazione della seconda regola la nostra agenda si svuota e il motore inferenziale si ferma

7.2 CLIPS

CLIPS e' un sistema a regole di produzione il cui componente principale sono i *fatti*.
I fatti possono essere

- Ordinati: dunque conta l'ordine degli elementi all'interno del fatto stesso
- Non ordinati: l'ordine non conta

Possiamo aggiungere un nuovo fatto alla nostra KB tramite l'istruzione
$assert(x)$
che aggiunge il fatto x alla KB. I fatti presenti nella KB vengono mostrati tramite il comando
$(facts)$

Esempio 1 : supponiamo di eseguire i due seguenti comandi in CLIPS

$(assert(lista\ a\ b\ c))$
$(facts)$

Il risultato che otteniamo e'
$f-0\ (initial-fact)$
$f-1\ (lista\ a\ b\ c)$

In CLIPS possiamo anche creare un template per costruire fatti piu' complessi. Tramite questo "stampino" potremo manovrare fatti complessi con maggiore semplicita'. Possiamo farlo tramite l'istruzione
$(deftemplate\ x)$

Esempio 2 : supponiamo di voler creare un template per dei fatti riguardanti persone. Il codice e' quello che segue

```
(deftemplate
   person "fatto non ordinato che rappresenta una persona"
      (slot name)
      (slot age)
      (slot eye-color)
      (slot hair-color)

)
(assert
   (person"fatto non ordinato che rappresenta una persona"
      (name "John Smith")
      (age 37)
      (eye-color black)
      (hair-color black)
   )
)
```

Notiamo che e' anche possibile dare dei fatti non necessariamente completi di tutti gli "attributi"

```
(assert
   (person
      (name "John Brown")
   )
)
```

Se proviamo a mostrare i fatti, abbiamo che

$f-0\ (initial-fact)$
$f-1\ (person\ (name\ "JohnBrown")\ (age\ nil)\ (eye-color\ nil)\ (hair-color\ nil))$

Quando il valore di un certo attributo non e' definito viene assegnato il valore di default *nil*

Possiamo anche modificare fatti preesistenti con la seguente sintassi:

$modify\ \#fatto\ (x\ v)$

che va a modificare l'attributo x del fatto numero $\#fatto$ assegnandoli il nuovo valore v.

Esempio 3 : supponiamo di scoprire l'eta' della persona John Brown di cui abbiamo parlato nell'esempio precedente. Se andassimo ad eseguire una modifica e invocassimo nuovamente i fatti contenuti nella KB come segue

$(modify\ 1\ (age\ 42))$
$(facts)$

otterremmo:
$f-0\ (initial-fact)$
$f-2\ (person\ (name\ "John\ Brown")\ (age\ 42)\ (eye-colornil)\ (hair-color\ nil))$

Notiamo una cosa importante: un fatto che viene modificato perde il numero che lo identifica diventando un fatto completamente nuovo. Infatti il fatto relativo a John Brown prima della modifica era il fatto numero 1. Dopo la modifica il fatto numero 1 viene rimosso e diventa il fatto numero 2.

Che un fatto modificato diventi un nuovo fatto e' una questione importante per via della rifrazione di cui abbiamo precedentemente parlato

Esempio 4 : supponiamo di voler aumentare lo stipendio di tutte le persone in un'azienda. La rifrazione impedisce che si formino loop in cui incrementiamo lo stipendio di una persona. Se pero' dovessimo andare a modificare

un fatto, per CLIPS otterremmo un nuovo fatto! E quindi il motore inferenziale di CLIPS sarebbe libero di applicare l'aumento di stipendio per la persona che abbiamo modificato visto che non e' piu' lo stesso fatto di prima

Un'altra istruzione in CLIPS e' la *duplicate* che va a duplicare un certo fatto. La sua sintassi e' la seguente:

$(duplicate\ \#fatto\ (x\ v))$

Tramite la duplicate andiamo a duplicare il fatto $\#fatto$ modificando il valore di tutti gli x attributi a cui assegniamo v

Esempio 5 : se riprendiamo l'esempio di prima ed eseguiamo

$duplicate\ 2\ (age\ 45))$
$(facts)$

Otteniamo:
$f-0\ (initial-fact)$
$f-2\ (person\ (name\ "John\ Brown")\ (age\ 42)\ (eye-color\ nil)\ (hair-color\ nil))$
$f-3\ (person\ (name\ "John\ Brown")\ (age\ 45)\ (eye-color\ nil)\ (hair-color\ nil))$

04/04/2017

Proseguiamo introducendo qualche elemento utile per programmare in CLIPS. Riconsideriamo l'esempio 2 in cui abbiamo definito un template per le persone. Purtroppo in CLIPS non e' possibile effettuare pattern matching sulle stringhe quindi, affinche' sia possibile ricercare una persona in base al suo nome dovremo andare a dare una definizione del template che non usi gli *slot* ma i *multislot* per tutti quei campi che coinvolgano una stringa.

Esempio 6 : andiamo a definire un nuovo template per le persone che ci permetta di eseguire pattern matching anche sulle stringhe. Il codice e' quello che segue

```
(deftemplate person2 "fatto non ordinato che rappresenta una
    persona"
   (multislot name)
   (slot age)
   (slot eye-color)
)
(assert(person2 (name "John Smith")(age 37)(eye-color black)))
(assert(person2 (name)(age 37)(eye-color black)))
```

Notiamo che il multislot lo avevamo gia' implicitamente usato quando avevamo creato una lista ordinata di elementi.

In CLIPS possiamo specificare un'insieme di regole in blocco tramite l'istruzione

$(deffacts\ name\ comment\ (f\ v))$

Con questa istruzione definiamo l'insieme di fatti avente nome *name* contenente la lista di fatti f aventi valore. Vediamo un esempio di definizione di fatti

Esempio 7 : possiamo definire l'insieme delle persone importanti

```
(deffacts popolazioneImportante "persone importanti"
   (assert(person2 (name "John Smith")(age 37)(eye-color black)))
)
```

Notiamo che andando ad eseguire la deffacts non vediamo subito i nuovi fatti nell'insieme. Per poterli "vedere" dobbiamo invocare prima una (*reset*) che elimina tutti i fatti precedentemente inseriti e aggiorna la memoria

Vediamo adesso come specificare un semplice programma CLIPS che usi delle regole.
Le regole sono definite tramite l'istruzione *defrule*

$(defrule\ name\ comment(precondizioni) => (azioni))$

affinche' una regola possa essere attivata devono valere le precondizioni. Vediamolo la defrule in un esempio

Esempio 8 : supponiamo di programmare un sistema che segnali le emergenze in un edificio. Programmiamo le emergenze dovute ad incendi e la risposta del nostro sistema esperto

```
(deftemplate emergency "emergenza incendio"
   (slot type))
(deftemplate response "risposta del sistema esperto"
   (slot action))
(defrule fire-emergency "in caso di incendio"
   ;;CONDIZIONI
   (emergency (type fire))
   =>
   ;;AZIONI
   (assert response (action activate-sprinkler-system))
)

;;FATTI
```

```
(deffacts emergenze
   (emergency (type flood))
)
```

Notiamo che all'inizio l'unico fatto che abbiamo e' che abbiamo un emergenza di alluvione e non di incendio. Dunque la regola relativa all'emergenza di incendio non puo' scattare.
Se dopo aver caricato il file nell'interprete CLIPS introduciamo un nuovo fatto

$(assert(emergency(type fire)))$

Allora vedremo subito che viene aggiunto all'insieme dei fatti la risposta del sistema, ovvero che e' necessario attivare il sistema antincendio.

05/04/2017

7.3 Pattern matching in CLIPS

Fino ad ora non abbiamo utilizzato variabili. Nell'ultimo esempio che abbiamo visto abbiamo dovuto definire un emergenza specifica per l'incendio. Come fare a implementare un'emergenza generica? Dobbiamo usare le variabili che vengono indicate con la sintassi $?nomeVariabile$. Vediamolo tramite il prossimo esempio

Esempio 1 : la nostra emergenza "generica" potrebbe essere definita come segue

```
(defrule emergencyRule "in caso di emergenza"
   (emergency (type ?e))
   =>
   (assert (alarm (type ?e)))
)
```

Notiamo che in corrispondenza di una certa emergenza, scatta l'allarme corrispondente. Infatti se introducessimo i due fatti

$(emergency\ (type\ flood))$
$(emergency\ (type\ fire))$

Allora l'allarme scatterebbe due volte

Esempio 2 : supponiamo di voler far scattare un allarme generale quando occorrono due emergenze. Il codice potrebbe essere

```
(defrule emergencyRule "in caso di doppia emergenza"
   (emergency (type ?e1))
```

```
   (emergency (type ?e2))
   =>
   (assert (alarm (type ?e)))
)
```

In realta' questo codice non e' corretto e l'allarme scatta gia' in presenza di una sola emergenza. Questo perche' non abbiamo stabilito che $e1$ ed $e2$ debbano essere diverse. Come in Prolog, dobbiamo aggiungere esplicitamente questo fatto. Andando ad effettuare un test sulla variabile come segue

$(test\ (neq\ ?e1\ ?e2))$

Fino ad ora abbiamo visto come sia possibile aggiungere nuovi fatti una volta che sono verificate alcune precondizioni. Vediamo come rimuovere alcuni fatti quando scatta una regola

Esempio 3 : vediamo come rimuovere alcuni fatti dalla nostra KB

```
(defrule fire-emergency "in caso di incendio"
   ?f <- (emergency (type ?e))
   (not (alarm (type ?e)))
   =>
   (assert (alarm (type ?e)))
   (retract ?f)
)
```

Notiamo che la variabile $?f$ viene introdotta nelle precondizioni per permetterci di eseguire la retract su di essa. Infatti dopo aver eseguito $?f <- (emergency\ (type\ ?e))$ in f e' contenuto il numero del fatto che verifica $(emergency\ (type\ ?e))$. A questo punto potremo riferire tale fatto nelle righe di codice successive.

In CLIPS possiamo anche utilizzare soltanto il simbolo ?
Usando solo il ? stiamo di fatto utilizzando una *wildcard* equivalente al simbolo di underscore utilizzato in Prolog.
Dobbiamo pero' porre attenzione al fatto che tramite il ? possiamo riferire solo fatti singoli. Se invece scriviamo $? allora riferiamo qualcosa che puo' essere formato da zero o piu' elementi. Vediamolo nell'esempio seguente

Esempio 4 : supponiamo di avere il seguente programma CLIPS

```
(defrule find-data1
   (data ? blue red $?)
   =>
   (assert (r1))
)
```

```
(deffacts data-facts
   (data 1.0 blue "red")
   (data 1 blue)
   (data 1 blue red)
   (data 1 blue RED)
   (data 1 blue red 6.9)
   (pippo)
   (data 2 red)
)
```

In questo caso solo il terzo e il quinto fatto farebbero scattare la regola $find - data1$

In CLIPS possiamo anche effettuare dei vincoli per enumerazione e concatenare proposizioni logiche nelle precondizioni da verificare

Esempio 5 : vediamo come effettuare una selezione per enumerazione

```
(defrule getperson2
   (person (age ?x&20|21|22) (name ?y))
   =>
   (assert (eta ?y ?x))
)
(deffacts people
   (person (name Joe) (age 20) ( father Bill))
   (person (name Bob) (age 20) (father Tom))
   (person (name Joe) (age 34) (father Luis))
   (person (name Sue) (age 34) (father Luis))
   (person (name Sue) (age 20) (father George))
   (person (name Bill) (age 54) (father Joseph))
   (person (name Joseph) (age 81)(father Bob))
)
```

In questo caso stiamo andando a ricercare il nome di tutte quelle persone che hanno 20, 21 o 22 anni.

Vediamo un altro esempio in cui usiamo gli operatori logici e mostriamo come stampare a schermo in CLIPS.

Esempio 6 : se volessimo definire una regola per stabilire se un numero sia maggiore di un altro, potremmo

```
(defrule greater-than
   (data ?y)
   (data ?x&:(> ?x ?y))
   =>
   (assert (greater ?x ?y))
)
```

Notiamo che non abbiamo usato una test come in precedenza anche se avremmo potuto abbiamo usato un altro modo.

Esempio 7 : presentiamo in questo esempio l'utilizzo dei predicati logici and e or

```
(defrule test-and
   (eta ?age&:(>= ?age 0)&:(< ?age 18))
   =>
   (printout t "ok: " ?age crlf)
)
(defrule test-or
   (eta ?age&:(< ?age 0) |: (>= ?age 18))
   =>
   (printout t "ko: " ?age crlf)
)
```

La prima regola trova tutte le persone minorenni, la seconda quelle maggiorenni. Notiamo invece che abbiamo usato l'istruzione printout che permette di stampare a schermo.

Notiamo infine che CLIPS non e' pensato per effettuare verifiche di goal come il Prolog. Infatti CLIPS dovrebbe applicare tutte le regole possibili e verificare se il goal negato fosse presente nella WM. Chiaramente procedere ad applicare tutte le regole in avanti fino a non poter piu' procedere e' una scelta abbastanza inefficiente. CLIPS e' invece pensato per inferire nuovi fatti.

06/04/2017

7.4 Salience e debugging

In CLIPS e' possibile dare una certa *priorita'* alle regole, anche detta *salience*. Questa priorita' puo' essere specificata all'interno di ogni regola, anche se non e' pensata per definire una gerarchia completa di regole. La salience viene invece utilizzata per dividere le regole in gruppi aventi priorita' diverse. La salience e' un valroe compreso tra -10.000 e +10.000

Esempio 1 : un semplice utilizzo della salience potrebbe essere quello di scrivere un programma CLIPS in modo simile al Prolog.

```
(defrule goal-sibling (declare (salience 10))
   (goal sibling ?x ?y)
   (or (sibling (s1 ?x) (s2 ?y))
       (sibling (s1 ?y) (s2 ?x)) )
   =>
   (printout t crlf ?x " is sibling of " ?y crlf)
)
```

Assegnando una salience superiore alla regola che verifica se abbiamo raggiunto il goal, andiamo ad ogni passo a testare se il goal sia stato raggiunto. Se invece vogliamo verificare che un certo goal non viene raggiunto, ci e' sufficiente specificare una regola con salience molto bassa che viene attivata solo quando non ci sono piu' regole applicabili.

```
(defrule goal-not-sibling
   (declare (salience -15))
   (goal sibling ?x ?y)
   (not (sibling (s1 ?x) (s2 ?y)))
   (not (sibling (s1 ?y) (s2 ?x)))
   =>
   (printout t crlf ?x " is NOT sibling of " ?y crlf) )

(defrule stop (declare (salience -100))
   =>
   (halt) )
```

In questo caso abbiamo definito due regole. La prima verifica che il goal non e' presente nella WM. La seconda tramite l'istruzione halt ferma l'inference engine.
Notiamo che l'insieme delle precondizioni della regola stop e' vuoto. Questo e' possibile perche' quando le precondizioni sono un'insieme vuoto, CLIPS procede a inserirvi automaticamente l'initial fact.
Per questo motivo l'initial fact non dovrebbe mai essere modificato.

Vediamo invece come fare un po' di debugging. In CLIPS possiamo specificare l'inserimento di un breakpoint tramite l'istruzione

set break nome_regola

che interrompe l'esecuzione ogni volta che attiviamo la regola $nome_regola$. Ovviamente possiamo anche rimuovere un break point con l'analoga istruzione

remove break nome_regola

Un altro modo per fare debugging consiste nel dare un argomento numerico all'istruzione run. Infatti specificando

run n

andiamo ad eseguire i primi n passi della computazione. Se nell'eseguire questi n passi incontriamo un break point, la computazione ovviamente si ferma.

Vediamo allora un esempio piu' strutturato che vada a mostrare come organizzare un po' meglio il codice in CLIPS. Nel corso dell'esempio useremo l'istruzione predefinita

bind res (op x y)

che esegue l'operazione *op* con operandi x e y andando a salvare il risultato in *res*.

Esempio 2 : supponiamo di voler realizzare un programma che si occupi di allocare e deallocare la memoria.
Un primo accorgimento da fare e' di suddividere la computazione in diverse fasi, secondo il paradigma *divide-et-impera*. Per questo motivo dividiamo il nostro programma in 3 fasi distinte

1. Accettazione di nuove richieste
2. Deallocazione
3. Allocazione

Nel nostro programma definiamo i template per la memoria, le applicazioni, le richieste di allocazione e deallocazione.
Il template piu' importante e' quello della memoria. La memoria possiede una propria dimensione, un certo spazio disponibile, e contiene un multislot con i nomi di tutte le applicazioni che hanno allocato qualcosa.
Mostriamo per esempio il codice da eseguire in fase di deallocazione

```
(deftemplate memory
   (slot size)
   (slot available)
   (multislot usage))
(deftemplate application
   (slot name)
   (slot mem-req))
(deftemplate allocate
   (slot name))
(deftemplate deallocate
   (slot name))

(defrule deallocation-ok
   ?ph <- (phase deallocation)
   (application (name ?x) (mem-req ?y))
   ?f1 <- (deallocate (name ?x))
   ?f <- (memory (available ?z) (usage $?prima ?x $?dopo))
   =>
   (bind ?m (+ ?z ?y))
   (modify ?f (available ?m) (usage $?prima $?dopo))
   (retract ?f1 ?ph)
   (assert (phase allocation))
)
```

Facciamo alcuni commenti: innanzitutto per poter deallocare dobbiamo essere sicuri di essere nella fase di deallocazione(prima riga della regola).

Le altre condizioni sono che deve esistere una applicazione che ha richiesto una deallocazione di memoria(seconda e terza riga). Inoltre dobbiamo anche verificare che quella stessa applicazione avesse allocato effettivamente qualcosa, ovvero che fosse nel multislot degli usage(quarta riga). Se tutte queste condizioni si verificano allora invochiamo l'istruzione bind che calcola la nuova quantita' di memoria disponibile e salva in risultato in m. A questo punto modifichiamo la memoria e andiamo a rimuovere due fatti: la fase corrente e la richiesta di deallocazione. Infine passiamo nella fase successiva asserendo il fatto phase allocation. Ovviamente onde evitare il blocco totale del programma dovremmo andare anche a specificare in un'altra regola una fase di deallocazione senza richieste di deallocazione.

Sebbene il programma del precedente esempio possa funzionare, dal punto di vista strutturale possiamo fare di meglio. Infatti in abbiamo mescolato due elementi tipici di un programma CLIPS. Ovvero il

- Domain knowledge: le informazioni sul dominio del problema
- Control knowledge: le informazioni sulla struttura del problema

Sarebbe molto meglio se riuscissimo a separare queste due entita' in regole diverse. Questo e' possibile a patto di ricordarsi di assegnare una salience maggiore alle regole che operano sul dominio e una salience minore a quelle che operano sul controllo.

Esempio 3 : riprendendo l'esempio di prima il problema principale e' che se vogliamo applicare delle modifiche strutturali, ad esempio allocare prima di deallocare, dobbiamo andare a modificare il codice in ogni singola fase. Supponiamo allora di scrivere una regola unica che gestisca le fasi come segue

```
(defrule change-phase
   (declare (salience -10))
   ?phase <- (phase ?current-phase)
   (phase-after ?current-phase ?next-phase)
   =>
   (retract ?phase)
   (assert (phase ?next-phase)))

(deffacts control-flow
   (phase accept-request)
   (phase-after accept-request deallocation)
   (phase-after deallocation allocation)
   (phase-after allocation accept-request))
```

Notiamo che la salience della regola di cambio fase e' bassa. Questo ci permette di eseguire prima tutte le operazioni sul dominio di una certa fase, e solo dopo di andare a cambiare fase.

7.5 Moduli

I moduli sono una componente CLIPS che ci serve per strutturare il codice. Vediamoli subito in un esempio

Esempio 1 : in CLIPS possiamo definire un modulo come segue

```
(defmodule M1)
(deftemplate M1::fatto (slot f1)...)
(deftemplate fatto (slot f1)...)        ;;default
```

La prima riga definisce che da quel punto in poi comincia il modulo M1, la seconda e la terza riga definiscono dei template per il modulo. Nello specifico la terza riga mostra come non sia necessario specificare ogni volta il modulo in cui siamo, questo viene determinato dall'interprete CLIPS.

In realta' avevamo gia' usato i moduli senza saperlo. Infatti di default viene ogni programma CLIPS comincia con l'istruzione che definisce il modulo *main*. Come abbiamo visto possiamo specificare fatti e regole all'interno di un modulo. Come potremmo aspettarci esiste uno scope per tali elementi. Infatti tutto quello che viene dichiarato all'interno di un modulo non e' visibile all'interno degli altri moduli.
Quello che possiamo fare e' rendere "pubblici" certi elementi del nostro programma CLIPS tramite le keywords *export* e *import*.

```
(defmodule MAIN (export ?ALL | ?NONE | deftemplate nomeTemplate))
;;in un altro modulo
(defmodule M1 (import MAIN ?ALL | ?NONE | deftemplate nomeTemplate))
```

Quando andiamo a specificare export/import nella definizione di un modulo possiamo esportare tutto, oppure niente oppure esportare solo alcuni template.

Strettamente legato ai moduli abbiamo il *focus*. Di fatto nel momento in cui andiamo ad applicare delle regole, stiamo applicando le regole di un preciso modulo. Se esistono piu' moduli che invocano gli uni le regole degli altri, allora viene a formarsi uno stack di moduli per stabilire quale sia il modulo corrente, ovvero quello di cui dobbiamo applicare le regole.
Questo ci permette di determinare in modo un po' piu' preciso quando la computazione termina. Infatti nel momento in cui non esistono piu' regole applicabili per un certo modulo, tale modulo viene rimosso dallo stack e passiamo ad applicare le regole del successivo. Se non vi sono piu' moduli sullo stack, la computazione si ferma.
Solitamente e' la normale esecuzione del programma che determina quale sia il modulo sul top dello stack. Esiste pero' un comando per cambiare manualmente

il modulo collocato sul top dello stack. Tale comando e' il *focus*.
Vediamolo subito in un esempio

Esempio 2 : vediamo il seguente programma CLIPS

```
;;NEL MAIN
(defrule r34
   ...
   =>
   (focus M1)
   (modify ...))

;;NEL MODULO M1
(defrule r45
   ...
   =>
   (retract ...)
   (pop-focus))
```

Quando nel main viene applicata la regola r34, nel caso in cui le precondizioni siano verificate viene invocato il comando focus M1.
Con tale comando spostiamo M1 sul top dello stack. Questo ci permettera' di provare ad applicare subito la regola r45.
Notiamo che dopo aver eseguito la regola r45 tramite il comando pop-focus che rimuove M1 dal top dello stack, spostiamo nuovamente il focus sul modulo main.

11/04/2017

8 Pianificare con CLIPS

La *pianificazione* e' il procedimento che porta alla determinazione di una serie di azioni che applicate allo stato iniziale, conducono ad uno stato finale.
Il procedimento di pianificazione deve essere portato a termine da un agente intelligente utilizzando la WM e la KB.
Chiaramente per poter pianificare dobbiamo avere a disposizione un *dominio di pianificazione*, ovvero un modello descrittivo di un ambiente che un pianificatore puo' utilizzare. Dunque abbiamo che

$$\Sigma = (S, A, \gamma)|(S, A, \gamma, c)$$

Dove S e' l'insieme finito degli stati, A l'insieme finito delle azioni. $\gamma : S \times A \to S$ e' la funzione di transizione da uno stato al successivo. Infine $c : S \times A \to [0, \infty]$ e' la funzione di costo. Quando la funzione di costo non e' definita esplicitamente, assegniamo ad ogni mossa costo unitario.
Faremo anche altre assunzioni nel seguito: l'ambiente e' statico e deterministico,

inoltre il tempo e' discreto e implicito: ovvero viene interpretato tramite una serie di stati.

Un'altra rappresentazione alternativa a quella appena vista e' la rappresentazione ispirata a STRIPS. In questo modello alternativo abbiamo che

$$U = \{B, R, F\}$$

Dove B e' l'insieme degli oggetti costruiti, R e' l'insieme delle proprieta'(se unarie) o relazioni(se non unarie) costanti, e definiscono l'ambiente. Infine F e' l'insieme dei fluenti $p(z_1, \ldots, z_n)$ dove z_i e' la variabile i-esima e nel complesso p puo' essere un letterale, una formula atomica, un atomo.

Esempio 1 : supponiamo di voler rappresentare un sistema di aeroporti e citta'. Allora:
$B = Citta' \bigcup Aeroporti \bigcup Oggetti$
dove $Citta' = \{losAngeles, boston\}$, e
$Aeroporti = \{aeroportoLosAngeles, aeroportoBoston\}$
$R = \{isIn(aeroportoBoston, Boston), isIn(aeroportoLosAngeles, losAngeles)\}$
$F = \{in(x, y)\}$ dove $x \in Oggetti$ e $y \in Aeroporti$.
Con quanto abbiamo appena definito possiamo specificare che un certo oggetto si trova presso un certo aeroporto.

12/04/2017

Vediamo come strutturare iterative deepening in CLIPS. Il codice sara' strutturato in cinque fasi distinte

1. Main: definisce i fatti lo stato iniziale e il goal. Determina una certa profondita' *maxdepth*. Stampa la soluzione
2. Expand: definisce i modelli delle azioni ed effettua il backtracking
3. Check: verifica se e' stata trovata una soluzione e la trasmette al main. Verifica la persistenza.
4. New: verifica se ci troviamo in uno stato nuovo
5. Del: rimuove lo stato appena aggiunto

Notiamo, che come gia' successo in Prolog, una volta determinato, iterative deepening puo' essere applicato a tutti i problemi a patto di modificare opportunamente il dominio su cui opera. Questo ci permette di risparmiare una buona parte di lavoro.
In questo caso, cercheremo di applicare iterative deepening al problema del mondo dei blocchi. Per farlo dobbiamo definire lo stato:

$$S_i = \{p_{i1} \ldots p_{in}\}$$

un esempio di stato nel mondo dei blocchi sara' qualcosa del tipo:
(*status i on a b*)
Per rappresentare che il blocco a è collocato sopra il blocco b.

Notiamo che in CLIPS e' particolarmente importante definire il backtracking correttamente. Infatti se in Prolog la ricerca all'indietro utilizzando la ricorsione ci facilitava di molto il lavoro, non possiamo fare altrettanto in CLIPS dove capita spesso di ritrovarsi in "vicoli ciechi" a causa della ricerca in avanti. Per questo dobbiamo sempre essere in grado di "tornare indietro". Per poter effettuare il backtracking dobbiamo usare la WM che tenga traccia degli stati, e l'agenda che ci ricorda le regole applicabili.
Vediamo ora il codice dell'iterative deepening applicato al mondo dei blocchi

Esempio 2 : come gia' detto il codice e' strutturato in cinque fasi. Ad ogni fase corrisponde un modulo

```
(defmodule MAIN (export ?ALL))
(deftemplate solution (slot value (default no)))
(deftemplate maxdepth (slot max))
(deffacts param
      (solution (value no))
      (maxdepth (max 0)))
(deffacts S0
     (status 0 clear a NA) (status 0 on a b ) (status 0 ontable b
         NA)
     (status 0 ontable c NA) (status 0 clear c NA)
     (status 0 handempty NA NA))
(defrule got-solution(declare (salience 100))
  (solution (value yes))
  (maxdepth (max ?n))
  =>
  (assert (stampa ?n)))
(defrule stampaSol(declare (salience 101))
  ?f<-(stampa ?n)
  (exec ?n ?k ?a ?b)
  =>
  (printout t " PASSO: "?n " " ?k " " ?a " " ?b crlf)
  (assert (stampa (- ?n 1)))
  (retract ?f))
(defrule stampaSol0(declare (salience 102))
  (stampa -1)
  => (halt))
(defrule no-solution (declare (salience -1))
  (solution (value no))
  (maxdepth (max ?d))
  =>
  (reset)
  (assert (resetted ?d)))
(defrule resetted
  ?f <- (resetted ?d)
  ?m <- (maxdepth (max ?))
  =>
```

```
   (modify ?m (max (+ ?d 1)))
    (printout t " fail with Maxdepth:" ?d crlf)
    (focus EXPAND)
    (retract ?f))

(defmodule EXPAND (import MAIN ?ALL) (export ?ALL))
(defrule backtrack-0 (declare (salience 10))
   ?f<- (apply ?s ? ? ?)
      (maxdepth (max ?d))
      (test (> ?s ?d))
   =>
   (retract ?f))
(defrule backtrack-1 (declare (salience 10))
   (apply ?s ? ? ?)
   (not (current ?))
   ?f1 <-  (status ?t&:(> ?t ?s) ? ? ?)
   =>
   (retract ?f1))
(defrule backtrack-2 (declare (salience 10))
   (apply ?s ? ? ?)
   (not (current ?))
   ?f2 <-  (exec ?t&:(>= ?t ?s) ? ? ?)
   =>
   (retract ?f2))
(defrule pick (declare (salience 2))
   (status ?s on ?x ?y)
   (status ?s clear ?x ?)
   (status ?s handempty NA NA)
   =>
   (assert (apply ?s pick ?x ?y)))
(defrule apply-pick3 (declare (salience 5))
   ?f <- (apply ?s pick ?x ?y)
   =>
   (retract ?f)
   (assert (delete (+ ?s 1) on ?x ?y))
   (assert (delete (+ ?s 1) clear ?x NA))
   (assert (delete (+ ?s 1) handempty NA NA))
   (assert (status (+ ?s 1) clear ?y NA))
   (assert (status (+ ?s 1) holding ?x NA))
   (assert (current ?s))
   (assert (news (+ ?s 1)))
   (assert (exec ?s pick ?x ?y )))
(defrule picktable (declare (salience 2))
   (status ?s ontable ?x ?)
   (status ?s clear ?x ?)
   (status ?s handempty NA NA)
   =>
   (assert (apply ?s picktable ?x NA)))
(defrule apply-picktable3 (declare (salience 2))
   ?f <- (apply ?s picktable ?x ?y)
```

```
   =>
   (retract ?f)
   (assert (delete (+ ?s 1) ontable ?x NA))
   (assert (delete (+ ?s 1) clear ?x NA))
   (assert (delete (+ ?s 1) handempty NA NA))
   (assert (status (+ ?s 1) holding ?x NA))
   (assert (current ?s))
   (assert (news (+ ?s 1)))
   (assert (exec ?s picktable ?x NA)))
(defrule put (declare (salience 2))
   (status ?s holding ?x ?)
   (status ?s clear ?y ?)
   =>
   (assert (apply ?s put ?x ?y)))
(defrule apply-put3 (declare (salience 5))
   ?f <- (apply ?s put ?x ?y)
   =>
   (retract ?f)
   (assert (delete (+ ?s 1) holding ?x NA))
   (assert (delete (+ ?s 1) clear ?y NA))
   (assert (status (+ ?s 1) on ?x ?y))
   (assert (status (+ ?s 1) clear ?x NA))
   (assert (status (+ ?s 1) handempty NA NA))
   (assert (current ?s))
   (assert (news (+ ?s 1)))
   (assert (exec ?s put ?x ?y)))
(defrule puttable (declare (salience 2))
   (status ?s holding ?x ?)
   =>
   (assert (apply ?s puttable ?x NA)))
(defrule apply-puttable3 (declare (salience 5))
   ?f <- (apply ?s puttable ?x ?y)
   =>
   (retract ?f)
   (assert (delete (+ ?s 1) holding ?x NA))
   (assert (status (+ ?s 1) ontable ?x NA))
   (assert (status (+ ?s 1) clear ?x NA))
   (assert (status (+ ?s 1) handempty NA NA))
   (assert (current ?s))
   (assert (news (+ ?s 1)))
   (assert (exec ?s puttable ?x NA)))
(defrule pass-to-check (declare (salience 25))
   (current ?s)
   =>
   (focus CHECK))

(defmodule CHECK (import EXPAND ?ALL) (export ?ALL))
(defrule persistency
    (declare (salience 100))
    (current ?s)
```

```
    (status ?s ?op ?x ?y)
    (not (delete ?t&:(eq ?t (+ ?s 1)) ?op ?x ?y))
   =>
   (assert (status (+ ?s 1) ?op ?x ?y)))

(defrule goal-not-yet
   (declare (salience 50))
   (news ?s)
   (goal ?op ?x ?y)
   (not (status ?s ?op ?x ?y))
   =>
   (assert (task go-on))
   (assert (ancestor (- ?s 1)))
   (focus NEW))
(defrule solution-exist
   ?f <- (solution (value no))
   =>
   (modify ?f (value yes))
   (pop-focus)
   (pop-focus))

(defmodule NEW (import CHECK ?ALL) (export ?ALL))
(defrule check-ancestor (declare (salience 50))
   ?f1 <- (ancestor ?a)
    (or (test (> ?a 0)) (test (= ?a 0)))
    (news ?s)
    (status ?s ?op ?x ?y)
    (not (status ?a ?op ?x ?y))
   =>
    (assert (ancestor (- ?a 1)))
    (retract ?f1)
    (assert (diff ?a)))
(defrule all-checked
   (declare (salience 25))
   (diff 0)
   ?f2 <- (news ?n)
   ?f3 <- (task go-on)
   =>
   (retract ?f2)
   (retract ?f3)
   (focus DEL))
(defrule already-exist
   ?f <- (task go-on)
   =>
   (retract ?f)
   (assert (remove newstate))
   (focus DEL))

(defmodule DEL (import NEW ?ALL))
(defrule del1(declare (salience 50))
```

```
   ?f <- (delete $?)
   => (retract ?f))
   (defrule del2
   (declare (salience 100))
   ?f <- (diff ?)
   =>
   (retract ?f))
(defrule del3(declare (salience 25))
   (remove newstate)
   (news ?n)
   ?f <- (status ?n ? ? ?)
   =>
   (retract ?f))
(defrule del4(declare (salience 10))
   ?f1 <- (remove newstate)
   ?f2 <- (news ?n)
   =>
   (retract ?f1)
   (retract ?f2))
(defrule done
   ?f <- (current ?x)
   =>
   (retract ?f)
   (pop-focus)
   (pop-focus)
   (pop-focus))
```

26/04/2017

8.1 Strategie di risoluzione dei conflitti

Abbiamo visto che è possibile utilizzare la salience per stabilire un'ordine di precedenza tra le regole. Cosa succede nel caso in cui si verifichi una parita' di salience? La priorità viene stabilita secondo i tre seguenti criteri

1. Salience
2. Applicando una strategia di conflict resolution
3. Applicando le regole secondo il loro ordine

Dunque a parita' di salience andiamo ad applicare una strategia di risoluzione dei conflitti e se anche la strategia non riesce a stabilire una priorita', allora si utilizza l'ordine delle regole.
Essendo il punti uno e tre gia' noti, vediamo quali sono le possibili strategie di risoluzione dei conflitti

1. depth: ha la precedenza la regola attivata dal fatto più recente

2. breadth: ha la precedenza la regola attivata dal fatto piu' vecchio

3. simplicity: ha la precedenza la regola che effettua meno confronti espliciti al suo interno

4. complexity: ha la precedenza la regola che effettua più confronti espliciti al suo interno

5. lex: si basa su un ordinamento lessicografico

   Esempio 1 : supponiamo di avere le seguenti regole:

   $r1: \ f1 \ f5$
   $r2: \ f5 \ f4$

   al suo interno, CLIPS riordina lessicograficamente i fatti relativi ad una certa regola come segue:

   $r2: \ f5, \ f4$
   $r1: \ f5, \ f1$

   Come possiamo notare i fatti vengono riordinati e visto che entrambe le regole possiedono il fatto $f5$ la priorità va alla $r2$ perché $f4 > f1$.

6. mea: variante del lex in cui i fatti non vengono riordinati, dopo si stabilisce un ordinamento lessicografico

   Esempio 2 : supponiamo di avere le seguenti regole

   $r1: \ f1, \ f5$
   $r2: \ f4 \ f5$
   $r3: \ f1, \ f3$

   In questo caso la strategia stabilisce il seguente ordine:

   $r2: \ f4 \ f5$
   $r1: \ f1, \ f5$
   $r3: \ f1, \ f3$

7. random: viene scelta una regola casualmente

Per specificare la strategia di risoluzione dobbiamo dare il comando

$(set - strategy \ nomeStrategia)$

04/05/2017

9 Reti Bayesiane

9.1 Incertezza e probabilità

Quando ci muoviamo in un ambiente di cui non abbiamo una totale conoscenza, non è possibile prevedere esattamente gli effetti di una certa azione. Cerchiamo di capirlo subito con un esempio

Esempio 1 : supponiamo di rappresentare l'azione "esco da casa per andare all'aeroporto t minuti prima che il volo parta" con il simbolo A_t.
Affermare che A_{25} mi permette di prendere l'aereo non è sempre vero. Questo è conseguenza del fatto che non ho una totale conoscenza dell'ambiente(non so se vi saranno incidenti lungo il percorso, molto traffico...). Partire con 24 ore di anticipo ci permette di essere ragionevolmente sicuri di raggiungere l'aereo, anche se non ne abbiamo ancora la certezza. E ad ogni modo risulta esagerato.

Diventa utile introdurre la *probabilità* che ci permette di associare ad ogni evento quali sono le possibilità di prevederne gli effetti.
Infatti la probabilità ci permette di riassumere gli effetti dell'ignoranza sull'ambiente.
In particolare la probabilità Bayesiana è quella che considera più eventi legati tra loro e ne considera la probabilità.

Per parlare di probabilità è necessario dividere tra la *probabilità a priori*(anche detta *probabilità incondizionata*) e la *probabilità condizionata*. La probabilità a priori e quella che viene utilizzata prima di essere a conoscenza di ulteriori informazioni.

Esempio 2 : supponiamo di avere le due seguenti variabili
Carie = {true, false}
Weather = {soleggiato, piovoso, nuvoloso, nevoso}
Supponiamo che:

$P(Carie = true) = 0.2, P(carie = false) = 0.8$
$P(Weather = soleggiato) = 0.72, P(Weather = piovoso) = 0.1,$
$P(Weather = nuvoloso) = 0.08, P(Weather = nevoso) = 0.1,$

Supponendo che i due fatti siano indipendenti, la probabilità a priori $P(Carie, Weather)$ è rappresentabile tramite una matrice che avrà dimensione pari a $\#Carie \times \#Weather$, ovvero:

Weather	soleggiato	piovoso	nuvoloso	nevoso
Carie = true	0.144	0.02	0.016	0.02
Carie = false	0.576	0.08	0.064	0.08

Se invece la probabilità è condizionata, le cose cambiano perche' esistono degli eventi legati fra loro. La probabilità condizionata è di solito espressa come segue:

$$P(a|b) = \frac{P(a \wedge b)}{P(b)}$$

Riprendendo l'esempio di prima potremmo notare che il mal di denti solitamente implica una carie, dunque $P(carie|malDiDenti) = 0.8$

Dalla formula di probabilità condizionata possiamo ricavare una regola più generale detta *chain rule*. Nello specifico abbiamo che:

$P(X_1, \ldots, X_n) = P(X_n|X_1, \ldots, X_{n-1})P(X_1, \ldots, X_{n-1}) = \ldots =$
$= \prod_{i=1}^{n} P(X_i|X_1, \ldots, X_{i-1})$

Notiamo che l'ordine delle variabili conta!
Un modo per calcolare le probabilità di eventi(indipendenti o meno) sarebbe quello per enumerazione che consiste nel compilare una tabella con tutte le probabilità, e andare a cercare le probabilità di eventi "atomici" vediamolo tramite un esempio

Esempio 3 : supponiamo di avere le tre variabili booleane
Carie = {true, false}
malDiDenti = {true, false}
presa = {true, false}
Dove la variabile presa indica se lo scovolino del dentista si incastra nel dente(fatto che se si verifica indica di solito la presenza di una carie). La tabella per enumerazione che potremmo costruire è la seguente

	malDiDenti		¬malDiDenti	
	presa	¬presa	presa	¬presa
carie	0.108	0.012	0.072	0.008
¬carie	0.016	0.064	0.144	0.576

Nota la tabella è abbastanza semplice calcolare le probabilità degli eventi. Ad esempio possiamo calcolare la probabilità di avere una carie sapendo che abbiamo un mal di denti:

$$P(carie|malDiDenti) = \frac{P(carie \wedge malDiDenti)}{P(malDiDenti)} =$$
$$= \frac{0.108 + 0.012}{0.108 + 0.012 + 0.016 + 0.064} = 0.6$$

Allo stesso modo possiamo calcolare la probabilità di non avere una carie sapendo che abbiamo un mal di denti:

$$P(\neg carie|malDiDenti) = \frac{P(\neg carie \wedge malDiDenti)}{P(malDiDenti)} =$$
$$= \frac{0.016 + 0.064}{0.108 + 0.012 + 0.016 + 0.064} = 0.4$$

Notiamo come, ovviamente, le due precedenti probabilità sommate diano valore 1. In particolare notiamo che il denominatore dei due precedenti calcoli è lo stesso! Allora può essere considerato una costante α e ad esempio:

$P(carie|malDiDenti) = \alpha P(carie \wedge malDiDenti) =$
$= \alpha[P(carie, malDiDenti, presa) + P(carie, malDiDenti, \neg presa)] =$
$= \alpha[< 0.108, 0.016 > + < 0.012, 0.064 >] = \alpha < 0.12, 0.08 > = < 0.6, 0.4 >$

Dove ad esempio $< 0.108, 0.016 >$ rappresenta le due probabilità in cui cambiamo il valore associato alla variabile carie ovvero:
0.108 è $P(carie, malDiDenti, presa)$ mentre 0.016 rappresenta
$P(\neg carie, malDiDenti, presa)$.

Notiamo una cosa importante: nel momento in cui otteniamo la coppia di valori $< 0.12, 0.08 >$ otteniamo una proporzione tra $P(carie|malDiDenti)$ e $P(\neg carie|malDiDenti)$. Per capire quale sia la probabilità effettiva di tali eventi ci è sufficiente normalizzare a 1.
Ma allora non abbiamo bisogno di conoscere $P(malDiDenti)$. Questo fatto è molto importante perché ci dice che per conoscere la probabilità di un generico $P(X|Y)$ e del suo complementare $P(\neg X|Y)$ non dobbiamo necessariamente conoscere $P(Y)$.

10/05/2017

Il problema dell'enumerazione è che è molto costosa in termini di spazio e tempo. Infatti supponendo di avere n variabili in cui quella con massima arietà di valori possieda d possibili valori, il costo in tempo e spazio è pari a $O(d^n)$.

Parliamo ora del concetto di indipendenza. Due variabili possono essere indipendenti in due modi diversi: *indipendenti* o *condizionalmente indipendenti*.

- Indipendenti: due variabili A e B sono dette indipendenti se vale che $P(A|B) = P(A)$ oppure se $P(B|A) = P(B)$ oppure se $P(A \wedge B) = P(A)P(B)$
- Condizionalmente indipendenti: consideriamo le tre variabili A, B, C. Diremo che A è condizionalmente indipendente da B dato C, se valgono le due seguenti equazioni:
  $P(A|B, C) = P(A|C)$
  $P(A|B, C) = P(A|\neg C)$

In generale l'indipendenza ci permette di ridurre il numero di variabili in gioco da esponenziale a lineare. In particolare l'indipendenza condizionale è lo strumento migliore che possiamo usare in ambienti di cui non abbiamo una totale conoscenza.

In base a quanto abbiamo detto, possiamo ricavare la legge di Bayes

$$P(a|b) = \frac{P(b|a)P(a)}{P(b)}$$

da cui possiamo facilmente ricavare che:

$$P(a|b)P(b) = P(b|a)P(a)$$

La legge di Bayes è particolarmente utile per determinare la probabilità di una causa dato un certo effetto. Vediamolo tramite un esempio

Esempio 4 : supponiamo di sapere che una persona contrae la meningite con probabilità pari a $P(men) = 10^{-4}$, che una persona abbia il mal di collo con probabilità pari a $P(coll) = 0.1$. Supponiamo infine che gli individui che hanno la meningite soffrano il mal di collo con probabilità $P(coll|men) = 0.8$.
Se una persona soffre di mal di collo, quale è la probabilità che abbia la meningite? La legge di Bayes ci dice che:

$$P(men|coll) = \frac{P(coll|men)P(men)}{P(coll)} = 8 \cdot 10^{-4}$$

15/05/2017

9.2 Sintassi e semantica delle reti Bayesiane

Visto che l'indipendenza(in particolare quella condizionale) ci permette di ridurre sensibilmente la complessità in tempo e spazio, vengono utilizzate le reti bayesiane il cui scopo è quello di rappresentare graficamente l'indipendenza condizionale.
Le sintassi delle reti bayesiane è la seguente

- Un insieme di nodi formato da un nodo per ogni variabile a cui viene associata una distribuzione condizionale
- La rete è di fatto un DAG[7]

Esempio 1 : riprendiamo l'esempio riguardante le carie, il mal di denti...
Una rete bayesiana adeguata potrebbe essere quella mostrata in Figura 14

Notiamo come il tempo atmosferico sia indipendente dalle altre variabili, mentre il mal di denti e la presa siano condizionalmente indipendenti tra loro data la carie.

Esempio 2 : supponiamo di ricevere una chiamata a lavoro dal nostro vicino John, il quale dice che l'allarme di casa sta suonando, ma l'altra vicina di casa, Mary, non sta chiamando. A volte l'allarme viene attivato da

[7] Directed Acyclic Graph

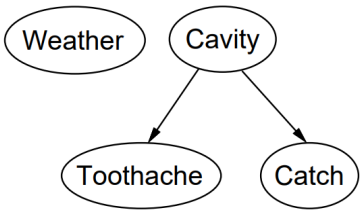

Figura 14: Rete bayesiana sul dominio del dentista

terremoti di entità minore. È ragionevole pensare che vi sia un ladro in casa?
In questo caso le variabili sono: Burglar(B), Earthquake(E), Alarm(A), JohnCalls(J), MaryCalls(M). Inoltre quello che sappiamo è che: un ladro può attivare l'allarme, un terremoto può attivare l'allarme, l'allarme può spingere John a chiamarci, l'allarme può spingere Mary a chiamarci. I dati a nostra disposizione sono i seguenti:

$P(B)$
0.001

$P(E)$
0.002

Inoltre sappiamo che

B	E	$P(A\|B,E)$
T	T	0.95
T	F	0.94
F	T	0.29
F	F	0.001

A	$P(J\|A)$
T	0.9
F	0.05

A	$P(M\|A)$
T	0.7
F	0.01

Allora la rete bayesiana potrebbe essere quella mostrata in Figura 15

Notiamo che in generale una tabella associata ad una certa variabile(dunque ad un certo nodo della rete) avente k nodi genitori, possiede 2^k righe.
Notiamo quindi che se ognuno degli n nodi della rete possiede al più k genitori, allora la rete necessita di $O(n * 2^k)$ valori per essere descritta. È importante notare che questo valore cresca linearmente con il numero dei nodi n, a differenza delle tecniche di enumerazione aventi costo $O(2^n)$. Nell'esempio appena visto, otteniamo 10 valori per descrivere la rete contro i $2^5 - 1 = 31$ valori che otterremmo tramite le tecniche di enumerazione.

In base a quanto detto finora, possiamo definire una *semantica globale* come segue:

$$P(x_1, \ldots, x_n) = \prod_{i=1}^{n} P(x_i | parents(X_i))$$

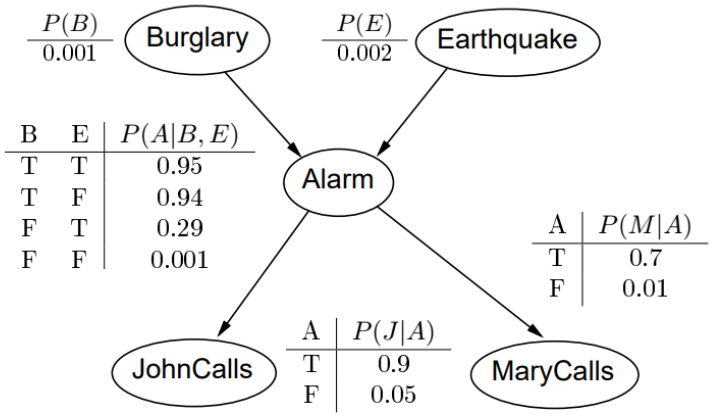

Figura 15: Rete bayesiana sul dominio del ladro

Data la rete bayesiana la regola di semantica globale ci permette di studiare i vari casi possibili

Esempio 3 : riprendendo l'esempio precedente, potremmo chiederci quale sia la probabilità che non vi sia un ladro, non vi sia un terremoto, ma che comunque l'allarme sia scattato e che abbiamo ricevuto una chiamata sia da John che da Mary, ovvero:

$P(j \wedge m \wedge a \wedge \neg b \wedge \neg e)$

La risposta può essere facilmente ottenuta tramite la regola di semantica globale appena descritta:

$P(j \wedge m \wedge a \wedge \neg b \wedge \neg e) = P(j|a)P(m|a)P(a|\neg b, \neg e)P(\neg b)P(\neg e) =$
$0.9 * 0.7 * 0.001 * 0.999 * 0.998 \simeq 6.28 * 10^{-4}$

Nelle reti bayesiane esiste anche il concetto di *semantica locale*: ogni nodo è condizionalmente indipendente da ogni suo nodo non discendente, dato il valore dei genitori. Ad esempio avevamo visto come nell'esempio del dentista il mal di denti e la presa fossero condizionalmente indipendenti tra loro data la carie.
In realtà possiamo anche dire che un certo nodo è condizionalmente indipendente da ogni altro nodo se siamo a conoscenza dei: genitori, figli e figli dei genitori. L'insieme di questi nodi viene detto *coperta di Markov*.

Come possiamo costruire una rete bayesiana dato un insieme di variabili $X_1, \ldots, X_n$? Esiste una tecnica basata su alcuni semplici passi:

1. Stabilire un ordinamento delle variabili $X_1, \ldots, X_n$

2. Per ogni i da 1 a n, aggiungere X_i alla rete selezionando un certo numero di genitori appartenenti a $X_1, \ldots, X_{i-1}$(ovvero posso segnare come genitori solo i nodi introdotti precedentemente) per cui vale che:
$P(X_i|parents(X_i)) = P(X_i|X_1, \ldots, X_{i-1})$

18/05/2017

9.3 Inferenza esatta

Il procedimento di inferenza alla luce delle reti bayesiane assume due nuove forme che vedremo nel seguito

- Enumerazione
- Eliminazione delle variabili

Il procedimento di enumerazione lo avevamo già visto nella sottosezione 9.1, ma dopo aver visto le reti bayesiane diventa per certi versi più efficiente

Esempio 1 : consideriamo nuovamente l'esempio del ladro della sezione precedente. Supponiamo di voler conoscere il valore di

$$P(B|j, m)$$

Normalmente,tramite l'enumerazione avremmo detto che:
$P(B|j, m) = \dfrac{P(B, j, m)}{P(j, m)} = \alpha P(B, j, m) = \alpha \sum_e \sum_a P(B, e, a, j, m)$
Ma alla luce delle reti bayesiane sappiamo alcune cose in più riguardo le dipendenze tra le variabili e dunque in base alla Figura 15 possiamo riscrivere la probabilità come segue:
$P(B|j, m) = \alpha \sum_e \sum_a P(B)P(e)P(a|B, e)P(j|a)P(m|a) =$
$= \alpha P(B) \sum_e P(e) \sum_a P(a|B, e)P(j|a)P(m|a)$

In questo caso il costo computazionale è pari a $O(n)$ in spazio e $O(d^n)$ in tempo dove d è la massima arietà e n il numero di nodi nella rete.

Il problema di questo approccio sta nella sua inefficienza dovuta alle computazioni ripetute. Analizziamo la complessità tramite l'albero di valutazione mostrato in Figura 16

Notiamo che per calcolare tutti i valori di e, l'albero di valutazione deve computare $P(j|a)$ e $P(m|a)$ per ogni valore di e.

Una soluzione alternativa all'inferenza esatta tramite enumerazione è l'inferenza tramite eliminazione delle variabili.
L'idea prende spunto dalla programmazione dinamica. Vediamolo riprendendo l'esempio precedente

Esempio 2 : supponiamo di voler computare nuovamente:

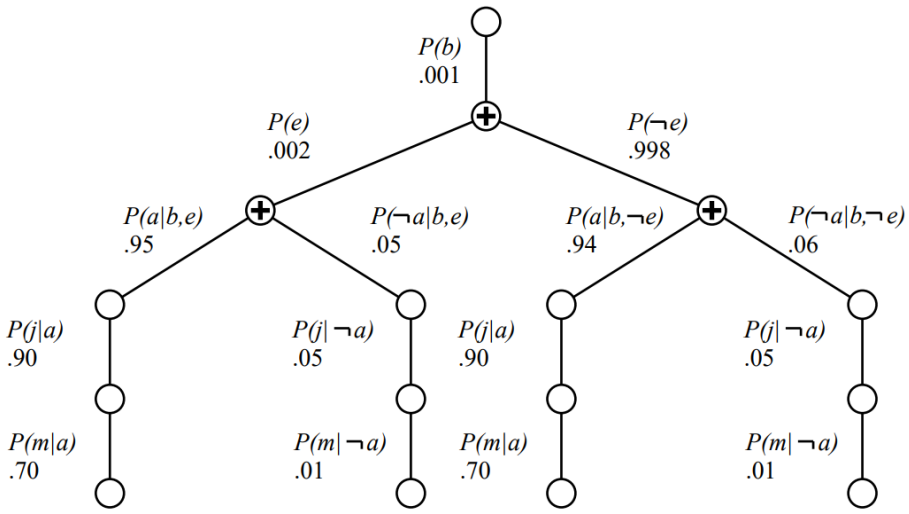

Figura 16: Albero di valutazione per il problema del ladro

$$P(B|j,m)$$

possiamo considerare l'intera computazione come segue effettuando una valutazione da destra verso sinistra, salvando i fattori passo passo evitando così le computazioni ripetute:
$P(B|j,m) = \alpha P(B)\sum_e P(e)\sum_a P(a|B,e)P(j|a)P(m|a) =$
$= \alpha P(B)\sum_e P(e)\sum_a P(a|B,e)P(j|a)f_M(a) =$
$= \alpha P(B)\sum_e P(e)\sum_a P(a|B,e)f_J(a)f_M(a) =$
$= \alpha P(B)\sum_e P(e)\sum_a f_A(a,b,e)f_J(a)f_M(a) =$
$= \alpha P(B)\sum_e P(e)f_{\bar{A}JM}(b,e) =$
$= \alpha P(B)f_{\bar{E}\bar{A}JM}(b) =$
$= \alpha f_B(b)f_{\bar{E}\bar{A}JM}(b)$

L'ultimo miglioramento che possiamo ottenere è quello dell'eliminazione delle variabili irrilevanti. Consideriamolo con l'esempio che segue

Esempio 3 : sempre considerando il dominio del ladro, vogliamo calcolare la seguente probabilità

$$P(JohnCalls|Burglary = true)$$

In questo caso possiamo scrivere che:
$P(J|b) = \alpha P(b)\sum_e P(e)\sum_a P(a|b,e)P(J|a)\sum_m P(m|a)$
Notiamo come la somma su tutti i valori di m sia pari a 1. Dunque M è irrilevante per il calcolo.

Più in generale, data una probabilità $P(X|E)$, diremo che una certa variabile Y è irrilevante a meno che non valga che

$$Y \in Antenati(\{X\} \cup E)$$

Nell'esempio visto poc'anzi
$X = JohnCalls$, $E = Burglary$, e $Antenati(X \cup E) = \{Alarm, Earthquake\}$
quindi la variabile MaryCalls è irrilevante.