Prolog: PROgramming in LOGic

- Il Prolog è un linguaggio Logico o Dichiarativo
- Utilizza solo i *fatti* e le *regole* per determinare la verità o la falsità di un *obiettivo* (goal).
- Il programmatore non specifica in nessun punto come risolvere il problema.
- Applicazioni in cui l'utilizzo del Prolog risulta vantaggioso: query su database, dimostrazione automatica di teoremi, realizzazione di parser, calcolo simbolico (per esempio: derivazione).

SWI Prolog

Sito da cui poter scaricare una shell Prolog

www.swi-prolog.org

Linguaggi logici

Un linguaggio logico è un insieme di *assiomi*, o *regole*, che definiscono delle relazioni tra oggetti. L'esecuzione di un programma logico non è altro che la derivazione di una conseguenza del programma. Gli assiomi sono espressi nel linguaggio logico tramite strutture dette *predicati*.

Un linguaggio logico si definisce del **1° ordine** se i predicati si riferiscono solo ad oggetti che a loro volta non sono ulteriormente definibili, ossia non sono a loro volta predicati.

Un linguaggio logico si definisce del **2° ordine** se i predicati si riferiscono anche a predicati che a loro volta si riferiscono ad oggetti non ulteriormente definibili.

È quindi possibile definire linguaggi di **ordine n** arbitrariamente elevato.

Struttura del linguaggio Prolog

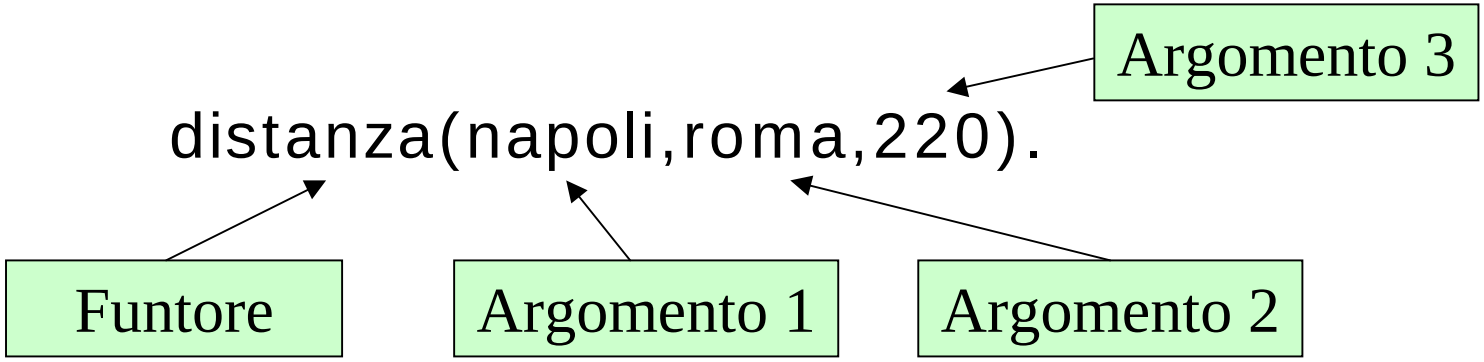
Il Prolog è un linguaggio formale del 1° ordine. In particolare è un linguaggio basato sulle **clausole di Horn**, ossia su un particolare sottoinsieme dei possibili predicati del primo ordine.

I predicati logici sono un modo semplice ed elegante per esprimere dei ragionamenti in termini logici.

Esempio di predicato logico e di suo significato:

distanza(napoli,roma,220). ←→ Napoli dista da Roma 220 Km

Predicato Logico (fatto)

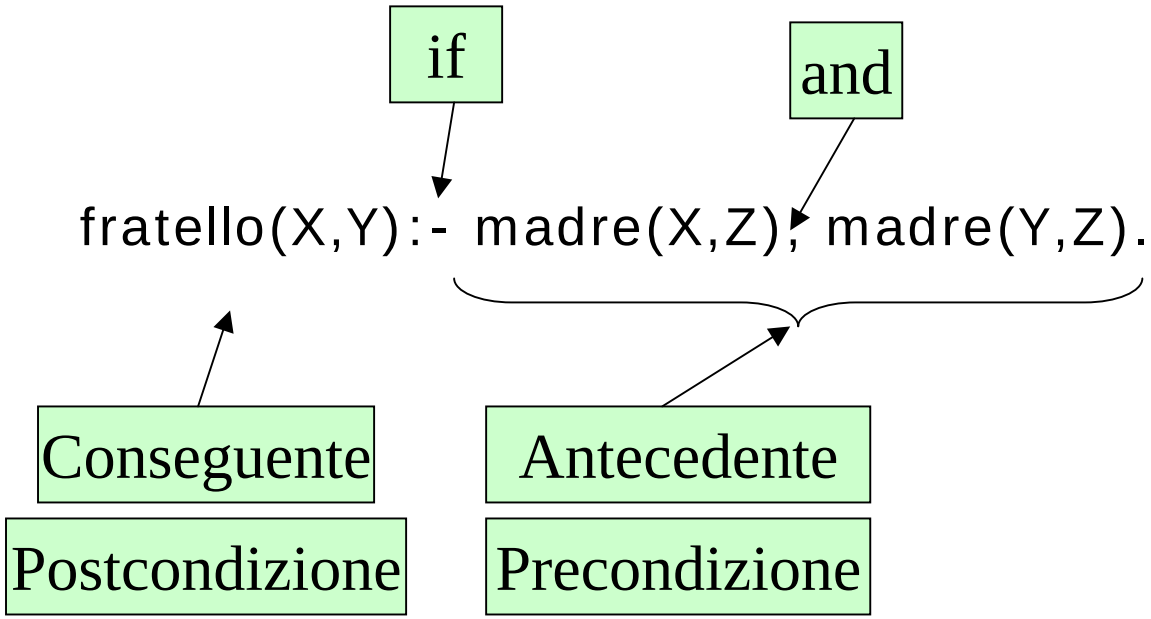


distanza(X,roma,220).

Significato: se esiste una città che dista 220 km da Roma, allora il predicato è vero. O meglio: Il predicato è vero se esiste una città X tale che Roma ne disti 220 km.

E' l'utente che stabilisce il significato dei fatti nel mondo reale, ossia che crea l'isomorfismo tra sistema reale e sistema formale.

Predicato Logico (regola)



Significato: X e Y sono fratelli se hanno la stessa madre Z.
O, formalmente: ∀X e ∀Y è vera la relazione di fratellanza tra X e Y se ∃Z madre di X e di Y.

Architettura di un Sistema Formale

Gli assiomi sono forniti al programma logico da un *esperto* del dominio di interesse, in quanto è necessario un conoscitore del problema che si vuole affrontare per stabilire quali sono le verità assiomatiche del caso.

Agli assiomi possono essere applicate delle regole, dette *regole di produzione* o *regole di inferenza* o *produzioni*, il cui risultato è generare nuovi fatti o *teoremi*.

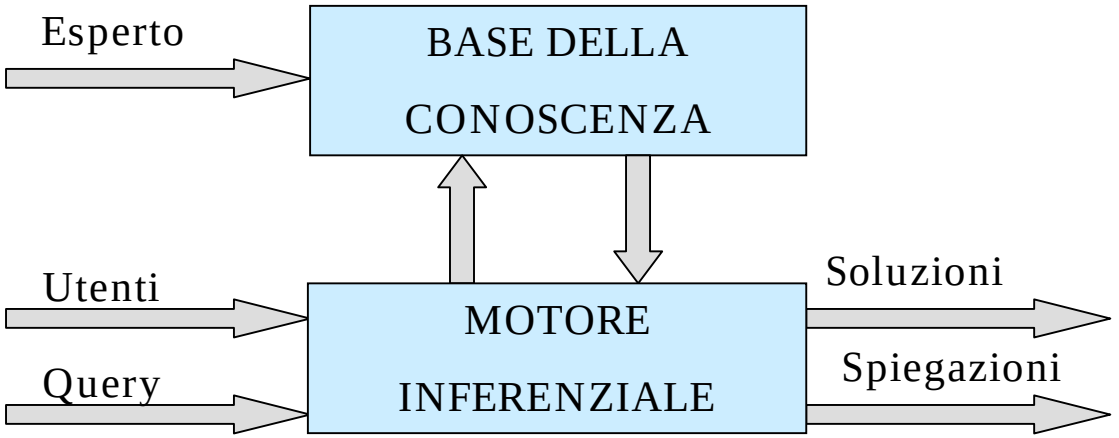
Quando viene generato un teorema questo viene incluso nella base della conoscenza, che, almeno in teoria cresce quindi dinamicamente. Il programma logico serve per dimostrare la verità o la falsità di un fissato teorema, il *goal*, solitamente non presente tra gli assiomi iniziali.

Architettura di un Sistema Formale

Il programma logico cerca il **goal** tra i fatti presenti nella base della conoscenza; se non lo trova applica una ad una le produzioni agli assiomi, ottenendo una serie di teoremi, che come già detto, vengono inclusi nella base della conoscenza.

Se il goal non è presente nella base della conoscenza aggiornata, il programma logico applica tutte le produzioni al nuovo insieme di fatti, generando altri teoremi, con un meccanismo che si ramifica sempre di più e che termina quando è stata dimostrata la verità o la falsità del goal (sistema *data driven*).

Architettura di un Sistema Formale



Il **motore inferenziale** del Prolog risponde affermativamente alle interrogazioni se è possibile ricavarne la risposta utilizzando i fatti e le regole che sono presenti nella **base della conoscenza (KB)**.

Architettura del Motore Inferenziale Prolog

Il motore inferenziale del Prolog tenta inizialmente di verificare il goal con uno degli assiomi della KB. Se non ci riesce, applica la prima produzione che incontra nella KB al goal. A questo punto il goal è stato modificato e si può tentare di verificare se tale nuovo goal verifichi uno degli assiomi della KB.

Se non è possibile verificare il goal, si può ancora una volta applicare ad esso una regola della KB e ritentare la verifica. Se il goal viene raggiunto, il procedimento di sostituzione termina, con l'asserzione della verità del goal, altrimenti, a valle di tutte le possibili applicazioni di tutte le regole si ha la falsità del goal. (sistema *goal driven*).

Clausole di Horn

Nell'ambito dei linguaggi di programmazione logica del primo ordine, un ruolo fondamentale è ricoperto dai linguaggi in cui le regole sono *clausole di Horn*. Una clausola di Horn è esprimibile nella forma:

$$\left.\begin{array}{l} A \leftarrow B_1,...,B_n \\ \\ A \leftarrow C_1,...,C_m \end{array}\right\} \text{con } n,m \geq 0$$

A è la *testa* della regola, ognuno dei $\boldsymbol{B_i}$ e dei $\boldsymbol{C_i}$ forma il *corpo* della regola; una possibile interpretazione della clausola è: A è vero sse $\forall$i è vero $\boldsymbol{B_i}$, oppure sse $\forall$j è vero $\boldsymbol{C_j}$. Quindi i $\boldsymbol{B_i}$ (ma anche i $\boldsymbol{C_j}$) sono in congiunzione tra loro, ma i due gruppi di letterali sono tra loro disgiunti.

Secondo questo formalismo gli assiomi (fatti) sono clausole che non hanno il corpo della regola, infatti la verità assiomatica non è implicata da nessuno, ma è da considerare sempre vera.

Variabili

Nelle clausole di Horn, gli argomenti dei predicati possono essere sia *costanti* che *variabili*.

Una variabile rappresenta un'entità non specificata e in Prolog non è tipata.

Le variabili si indicano convenzionalmente con Lettera Maiuscola, mentre le costanti con lettera minuscola.

Una variabile è libera se occorre in un solo predicato della clausola: in tal caso assume un valore, ma questo valore sarà gettato via, senza mai essere utilizzato.

Il Prolog consente di utilizzare variabili anonime, ossia variabili che non vengono realmente istanziate. Le variabili anonime vengono indicate da stringhe che iniziano con underscore (_)

Sostituzioni e Istanze

Se un argomento di un fatto è una variabile, p.es `padre(X,Y)` (il cui significato che si dà è: X è il padre di Y), allora una **sostituzione** consiste nell'inserire una costante al posto di una variabile.

In generale si dirà che il predicato B è un'**istanza** di A se B è stato ottenuto da A tramite una sostituzione.

Esempio:

considerato il fatto

`padre(marco,Y)` (marco è padre di Y)

possibili istanze sono:

`padre(marco,andrea)` (marco è padre di andrea)

`padre(marco,mario)` (marco è padre di mario)

Unificazione

Un'**unificazione** di due termini è una sostituzione che li rende identici. Per esempio la sostituzione *Y* = *andrea* unifica i due predicati

```
padre(marco,Y).
padre(marco,andrea).
```

In generale è vero che:

due costanti unificano se sono uguali;

una costante ed una variabile unificano sempre e la variabile assume il valore della costante;

due variabili unificano sempre;

Comandi dell'ambiente Prolog

Una Knowledge Base è editata in un file di testo nominato come **KBname.pl** La KB viene acquisita (compilata) dalla shell Prolog tramite il comando:

```
consult(KBname).
```

Quando viene richiesta la verifica di un predicato al motore inferenziale, la ricerca si ferma al primo **goal** trovato. Con il comando '`;`' si richiede al motore inferenziale di cercare un ulteriore goal.
Con il comando '`a`' si sospende l'esecuzione della ricerca

Comandi dell’ambiente Prolog

Per verificare uno ad uno tutti i passi seguiti dal motore inferenziale per tentare di unificare il goal, si può attivare utilizzare il comando

```
trace.
```

Tale comando va utilizzato subito prima di richiedere la verifica di un predicato. E’ presente anche una versione grafica del comando trace.

Un primo esempio: fatti

La base della conoscenza è costituita da una serie di fatti e di regole.

Nei fatti è specificato il sesso di ognuna delle persone di una famiglia e le relazioni madre-figlio e padre-figlio.

```
maschio(renato).
maschio(corrado).
maschio(dario).
maschio(corradosr).
maschio(fernando).
maschio(corrado2).
maschio(francesco).
maschio(giuseppe).
maschio(antonio).
```

```
padre(renato,corrado).
padre(renato,dario).
padre(corradosr,renato).
padre(corradosr,francesco).
padre(corradosr,mariella).
padre(fernando,tina2).
padre(antonio,giuseppe).
padre(antonio,laura).
padre(francesco,corrado2).
padre(francesco,manuela).
```

```
femmina(tina).
femmina(franca).
femmina(lucia).
femmina(mariella).
femmina(laura).
femmina(manuela).
femmina(tina2).
femmina(paola).
```

```
madre(tina,franca).
madre(tina2,corrado).
madre(tina2,dario).
madre(lucia,renato).
madre(lucia,francesco).
madre(lucia,mariella).
madre(franca,tina2).
madre(mariella,giuseppe).
madre(mariella,laura).
madre(paola,corrado2).
madre(paola,manuela).
```

Un primo esempio: regole

Le regole servono invece per definire relazioni parentali sulla base dei fatti presenti nella KB.
Per esempio, un genitore è o un padre o una madre e questo può essere espresso dalla regola

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

Cioè: X è genitore di Y se X è padre di Y oppure se X è madre di Y.

Un primo esempio: regole

Altra regola: Y è figlio di X se X è padre di Y ed Y è maschio oppure se X è madre di Y ed Y è maschio.

```
figlio(Y,X) :-
   padre(X,Y),maschio(Y).
figlio(Y,X) :-
   madre(X,Y),maschio(Y).
```

Un primo esempio: regole

Una regola analoga può essere espressa per la figlia: Y è figlia di X se X è padre di Y ed Y è femmina oppure se X è madre di Y ed Y è femmina.

```
figlia(Y,X) :-
   padre(X,Y),femmina(Y).
figlia(Y,X) :-
   madre(X,Y),femmina(Y).
```

Un primo esempio: regole

Due persone sono fratelli se hanno un genitore in comune.

```
fratello(X,Y):-
        genitore(Z,X),
        genitore(Z,Y),
        maschio(X),
        not(equal(X,Y)).
```

Il problema è che se X e Y sono uguali, il predicato è vero, quindi ogni persona risulta anche fratello di se stesso. Per ovviare a questa imprecisione, bisogna introdurre un predicato di uguaglianza:

```
equal(X,X).
```

Un primo esempio: regole

```
zio(X,Y):-
   fratello(X,Z),
   padre(Z,Y),
   maschio(X).
zio(X,Y):-
   fratello(X,Z),
   madre(Z,Y),
   maschio(X).
```

X è zio di Y se esiste una persona Z
tale che X è fratello di Z,
Z è padre (oppure madre) di Y
ed X è maschio

Un primo esempio: regola ricorsiva

Si consideri ora il caso di una relazione più elastica, quella di antenato. Una persona è un antenato di un altra, se ne è il genitore, oppure il nonno oppure il bisnonno e così via. E' comodo esprimere questa relazione con una sola clausola. Tale clausola deve essere ricorsiva.

Il caso base, ossia il caso che può essere verificato in maniera immediata, è quello in cui si verifica se una persona è genitore di un altra.

```
antenato(X,Y):-
    genitore(X,Y).
```

Un primo esempio: regola ricorsiva

In generale X è antenato di Y se esiste una persona Z tale che X è genitore di Z ed a sua volta Z è antenato di Y.

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

Variabili Singleton

Si consideri la frase: **un cibo è qualsiasi cosa qualcuno mangi**. Tale frase può essere espressa tramite il seguente predicato:

```
cibo(X) :- mangia(Y,X).
```

Tale predicato, una volta consultata la KB, dà luogo ad un warning

{Warning: [Y] - singleton variables in cibo/1 in lines 3-4}

Questo significa che, nel tentativo di soddisfare il goal, alla variabile Y sarà assegnato un valore che non verrà mai utilizzato.

Una variabile ‘singleton’ è una variabile che occorre soltanto in un predicato di una clausola: assume un valore dopodiché tale valore viene buttato via.

Per eliminare il warning, ossia per indicare che non interessa sapere qual è l’unificazione della variabile Y, si può riscrivere la clausola come:

```
cibo(X) :- mangia(_Y,X).
```

Quantificazione delle variabili

	Quantificazione esistenziale	**Quantificazione universale**
FATTI	Mai	Sempre
REGOLE	Variabili che compaiono almeno nella precondizione	Variabili che compaiono solo nella postcondizione
QUERY	Sempre	Mai

Quantificazione delle variabili: un esempio

```
nonno(X,Z) :- padre(X,Y), padre(Y,Z).
```

Tale clausola può essere letta:

$\forall$X e $\forall$Z, X è il nonno di Z se $\exists$Y tale che Y è padre di X e Z è padre di Y.

Si vede che la quantificazione esistenziale è associata all'unica variabile che compare solo nella precondizione.

Motore Inferenziale: Unificazione Goal Driven

Il Prolog utilizza un approccio **goal driven**:
Il primo passo è tentare l'unificazione del goal con uno degli assiomi della KB.
Se tale unificazione non riesce, allora si applica una delle regole al goal. A questo punto il goal è cambiato ed il MI tenta di unificare il nuovo goal, cioè il *goal corrente*, con uno degli assiomi della KB.
Se l'unificazione non riesce, verrà cambiato il goal corrente tramite l'applicazione di un'altra regola presente nella KB. Il procedimento si sospende o alla prima unificazione oppure quando si è tentata, senza successo l'applicazione di tutte le possibili regole presenti nella KB.

Motore Inferenziale: Ricerca Depth First

Il MI tenta di unificare il goal procedendo in *profondità* lungo l'albero di ricerca, percorrendo un suo ramo finché o trova una soluzione, o raggiunge uno stato senza successori.

Al fine di esplorare tutto l'albero di ricerca, il MI deve essere in grado, in qualunque nodo si trovi, di risalire di un livello e ridiscendere lungo un altro ramo. Tale operazione è chiamata *backtracking* ed è una componente della ricerca in profondità.

Un primo esempio: query

Una volta che popolata la base di conoscenza con fatti e regole, è possibile effettuare delle query.

Una query è espressa tramite un predicato o un AND di predicati.

Il motore inferenziale del Prolog, consultando la base di conoscenza, cercherà di unificare il goal con una delle regole. Se l'unificazione riesce, la risposta sarà affermativa, altrimenti sarà negativa.

Esempio:

```
?- fratello(dario,corrado).
Yes
```

Un primo esempio: query

```
?- fratello(renato,X).
X = francesco ;
X = mariella ;
X = francesco ;
X = mariella ;
No
```

Viene chiesto al motore inferenziale chi siano i fratelli di renato.

Il sistema trova una prima unificazione del goal con una delle regole.

Con il comando ‘;’ si richiede di cercare successive unificazioni.

Ne esiste una seconda, dopodiché viene riproposta nuovamente la prima unificazione.

Questo è dovuto al fatto che si raggiunge lo stesso goal utilizzando regole diverse.

Un primo esempio: query

```
?- zio(X,Y).

X = renato
Y = corrado2;

X = renato
Y = manuela;

X = francesco
Y = corrado;
...
```

Viene chiesto al motore inferenziale di mostrare tutte le relazioni in cui una persona è zio di un'altra.

Un primo esempio: query composta

Viene chiesto al motore inferenziale di trovare tutte le sorelle degli zii di corrado.

```
?- sorella(X,Y),zio(Y,corrado).

X = mariella
Y = francesco ;
```

Comando TRACE

Il TRACE serve a visualizzare uno ad uno tutti i passi seguiti dal MI. per determinare l'unificazione del goal.

Il comando `trace.` attiva la visualizzazione. Ad ogni passo viene visualizzato il goal corrente, il quale viene modificato in conseguenza delle regole di inferenza.

Il MI è in una delle tre fasi nel seguito elencato:

call: si tenta per la prima volta l'unificazione di un nuovo goal con una delle regole sistema.

fail: se il goal corrente non viene unificato con la regola selezionata, il sistema 'esce' da questo goal, ossia il tentativo di unificazione termina. Al fail segue una fase di backtracking.

redo: il MI tenta nuovamente di unificare un goal già provato almeno una volta con una regola diversa.

Derivatore simbolico: Fatti

Si vuol realizzare un programma che calcoli la derivata di una funzione di una variabile.

Il primo passo è, come nell'esempio precedente, la costruzione della KB inserendo i fatti. In questo caso i fatti sono le regole di derivazione.

```
d(sen(X),cos(X),X).
d(cos(X),-sen(X),X).
d(X,1,X).
```

Derivatore simbolico: Regole

```
d(F+G,H+I,X):-
    d(F,H,X),
    d(G,I,X).
```

Derivata della somma

```
d(F-G,H-I,X):-
    d(F,H,X),
    d(G,I,X).
```

Derivata della differenza

```
d(F*G,H*G+F*I,X):-
    d(F,H,X),
    d(G,I,X).
```

Derivata del prodotto

```
d(F/G,(H*G-F*I)/G*G,X):-
    d(F,H,X),
    d(G,I,X).
```

Derivata del rapporto

Derivatore simbolico - query

```
?- d(sen(x),G,x).
G = cos(x);
No
```

Scelto x come nome per la variabile X, esiste una funzione G tale che sen(x) ne sia la derivata?

```
?- d(sen(X),G,X).
X = _G157
F = cos(_G157) ;
```

Esiste una variabile tale che esiste una funzione G tale che il seno di quella variabile ne sia la derivata?

Derivatore simbolico - query

```
?- d(sen(X),cos(Y),Z).

X = _G160
Y = _G160
Z = _G160
```

Sotto quali ipotesi si ha che il coseno di una variabile è il seno di un'altra ?

Derivatore simbolico - query

```
?- d(Y,cos(x),x).

Y = sen(x) ;

No
```

Esiste una funzione Y tale che cos(x) ne sia la derivata rispetto alla variabile x? O, in altri termini, esiste una primitiva di cos(x)?

Liste

Le *liste* in Prolog sono un tipo predefinito. Una lista può contenere elementi di qualsiasi tipo, in altri termini gli elementi non sono tipati. Una lista è delimitata da parentesi quadre. (lista chiusa)

Esempi: **[1,2,tre,4]** **[a,b,[2,c,3],3].** (lista chiusa)

E' anche possibile esprimere le liste in termini di testa e coda. La testa è il primo elemento mentre la coda è costituita da tutti gli altri elementi della lista. Per separare la testa dalla coda si usa il separatore '|'.

Esempio: **[X|Y].** (lista aperta)

Liste: Unificazione

Una **costante** non unifica con una **lista chiusa**.

a ←→ **[1,x,3,q,…]** NO

Una **variabile** unifica con una **lista chiusa**.

X ←→ **[1,x,3,q,…]** SI: la variabile assume come valore l'intera lista.

L'unificazione è X = [1,x,3,q,…]

Liste: Unificazione

Due **liste chiuse** unificano se:

1) hanno la stessa cardinalità;
2) gli elementi unificano ordinatamente;

[napoli,roma,X,venezia,4] ←→ [napoli,Y,genova,Z,4] **SI**
L'unificazione è: **Y = roma; X = genova; Z = venezia;**

[napoli,roma,X,venezia,4] ←→ [napoli,X,genova,Z,4] **NO**
in quanto X dovrebbe assumere contemporaneamente i valori
X = roma ; X = genova.

Liste: Unificazione

Una **lista aperta** ed una **lista chiusa** unificano se esiste un'unificazione tra la testa della lista chiusa ed il primo elemento della lista aperta e la coda della lista chiusa ed i rimanenti elementi della lista aperta.

lista1	**lista2**	**unificazione**
[cane,gatto,topo]	**[X|Y]**	X=cane, Y=[gatto,topo]
[cane,gatto,topo]	**[X,Y|Z]**	X=cane, Y=gatto, Z=[topo]
[[cane,gatto],topo]	**[X,Y|Z]**	X=[cane,gatto], Y=topo,Z=[]
[3]	**[X|Y]**	X=3, Y = []
[3]	**[X,Y|Z]**	non c'è unificazione

Principio di Induzione Matematica

Sia data una successione x_0, x_1, … x_n, … di infiniti termini ed una proprietà P.

Si vuol dimostrare che la proprietà P vale per ogni elemento della successione. Si tratta in teoria di effettuare infinite dimostrazioni.

Il principio di induzione afferma che, se è dimostrato che la proprietà P vale per l'elemento x_0 (caso base) e se, ipotizzando che la proprietà P è vera per un generico termine x_i, si riesce a dimostrare che lo è anche per x_{i+1}, allora la proprietà P è vera per ogni termine della successione.

$P(x_0)$ è vera

hp: $P(x_i)$ vera

th: $P(x_{i+1})$ vera

$\forall j$ $P(x_j)$ vera

Liste – Appartenenza

Per poter organizzare programmi logici efficienti sulle liste si può utilizzare la rappresentazione [Testa|Coda].

Per costruire un predicato che verifichi se un elemento è membro di una lista, bisogna verificare due possibili casi:

1) l'elemento è la testa della lista

2) l'elemento è membro della coda della lista.

Il caso 1 viene banalmente rappresentato dalla clausola:

```
membro(A,[A|B]).
```

Liste – Appartenenza

Il caso 2 può essere risolto in maniera ricorsiva:
Un elemento A è membro di una lista [H|C] se è membro della sua coda C.

```
membro(A,[H|C]):-
        membro(A,C).
```

Il problema dell'appartenenza ad una lista è ricondotto quindi alla verifica dell'appartenenza dell'elemento ad una sottolista della lista di partenza.

Liste – Appartenenza

Esempi:

```
?- membro(1,[1,2,3]).
Yes
```

1 è membro della lista [1,2,3] ?

```
?- membro(X,[1,2,3]).
X = 1 ;
X = 2 ;
X = 3 ;
No
```

Quali sono i membri della lista [1,2,3] ?

Liste – Lunghezza

La lunghezza di una lista può essere determinata in maniera efficiente decomponendo il problema in due sottoproblemi:

1) la lunghezza di una lista vuota è 0.

2) la lunghezza di una lista è 1 più la lunghezza della sua coda

Il problema 1 è di soluzione immediata, mentre con il problema due si riconduce la determinazione della lunghezza della lista alla determinazione della lunghezza di una sottolista della lista di partenza.

Liste – Lunghezza

```
lung([],0).

lung([Testa|Coda],N):-
   lung(Coda,NUM1),
   plus(NUM1,1,N).
```

Esempio

```
?- lung([1,2,3,4,5],X).
 X = 5;
No
```

Liste – Accodamento

Per costruire un predicato che concateni una lista in coda ad un'altra lista, bisogna verificare due possibili casi:
1) la lista alla quale si vuol concatenare la seconda lista è vuota
2) si accoda la seconda lista alla sottolista che si ottiene eliminando la testa della lista di partenza.

Il caso 1 viene descritto dalla clausola:

```
appendi([],Lista2,Lista2).
```

Cioè, inserendo una lista in coda ad una lista vuota, si ottiene proprio la lista inserita.

Liste – Accodamento

Se la lista di partenza non è vuota, allora l'inserimento può essere effettuato sulla sottolista che si ottiene escludendo la testa della lista di partenza.

```
appendi([Testa1|Coda1],Lista2,[Testa1|Coda3]):-
        appendi(Coda1,Lista2,Coda3).
```

Ancora una volta, il problema viene affrontato riconducendolo ad un problema della stessa natura ma di dimensioni più ridotte.

Liste – Accodamento

Esempi:

```
?- appendi([1,2,3],[4],X).
X = [1, 2, 3, 4] ;
```

Quale lista si ottiene accodando la lista [4] alla lista [1,2,3] ?

```
?- appendi([1,2,3],X,[1,2,3,4]).
X = [4] ;
```

Quale lista X si deve accodare alla lista [1,2,3] per ottenere la lista [1,2,3,4] ?

Liste – Accodamento

```
?- appendi(X,Y,[1,2,3,4]).

X = []
Y = [1, 2, 3, 4] ;

X = [1]
Y = [2, 3, 4] ;

X = [1, 2]
Y = [3, 4] ;

X = [1, 2, 3]
Y = [4] ;

X = [1, 2, 3, 4]
Y = [] ;
```

Quali sono le liste X, Y tali che accodando la lista Y alla lista X si ottiene la lista [1,2,3,4] ?

Liste – Inversione

Per costruire un predicato che inverta una lista, ancora una volta si può utilizzare il metodo di decomposizione in sottoproblemi.
E' quindi necessario individuare un caso di soluzione immediata (caso base) ed una modalità per ricondurre l'inversione di una lista all'inversione di una sottolista della lista di partenza. Il caso base è:

```
inverti([],[]).
```

ossia, l'inversione di una lista vuota è ancora una lista vuota.

Liste – Inversione

Nel caso in cui la lista di partenza non sia vuota, la si può rappresentare nella forma [Testa|Coda], invertire la coda e poi, invertita la coda, le si può concatenate la testa.

```
inverti([T1|C1],L2):-
        inverti(C1,X),
        appendi(X,[T1],L2).
```

Questa clausola opera però in maniera inefficiente, in quanto per una lista di lunghezza n, viene richiamato n volte il predicato di concatenazione, il quale ha una complessità O(n). La complessità è quindi $O(n^2)$.

Liste – Inversione

E’ possibile pensare ad una formulazione più efficiente dell’inversione, utilizzando una lista di appoggio.

Il caso base consiste nell’avere una lista vuota ed una lista d’appoggio già invertita.

```
inverti([],Lista2,Lista2).
```

La lista invertita è, in tal caso, esattamente la lista d’appoggio.

Liste – Inversione

Ricordiamo che l'accumulatore contiene una lista già invertita. L'inversione della lista può in questo caso essere ricondotta all'inversione di una sottolista. Più in dettaglio, si estrae la testa della lista di partenza, e la si inserisce in testa alla lista accumulatore.

```
inverti([Testa1|Coda1],Acc,Lista2):-
        inverti(Coda1,[Testa1|Acc],Lista2).
```

Liste – Inversione

Inoltre l'utente finale non è interessato a vedere la presenza dell'accumulatore, il quale può essere agevolmente nascosto con la clausola:

```
inverti(X,Y) :-
        inverti(X,[],Y).
```

rendendo così visibile all'esterno un predicato con due argomenti.

Liste – Inversione

Il programma logico completo è:

```
inverti(X,Y) :-
   inverti(X,[],Y).

inverti([],L2,L2).

inverti([Testa1|Coda1],Acc,Lista2):-
   inverti(Coda1,[Testa1|Acc],Lista2).
```

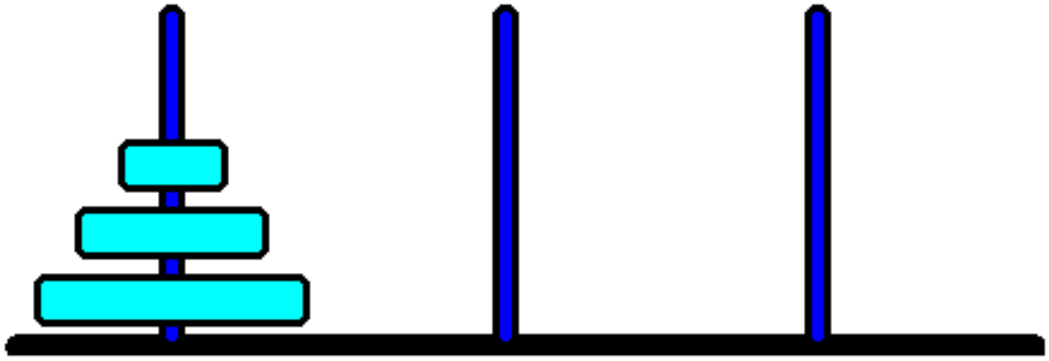
Torri di Hanoi

Scopo del gioco:

Spostare i dischi dal piolo di sinistra al piolo di destra.

Regole:

- Un disco più grande può essere spostato solo su un disco più piccolo.
- Si può spostare un solo disco alla volta e solo il disco in cima alla pila.



Torri di Hanoi – caso base

Il caso base, cioè che può essere risolto in maniera immediata, è quello in cui si ha un solo disco: esso può essere infatti spostato direttamente dal piolo sinistro a quello destro, in un sol passo.

E' quindi possibile risolvere tale caso mediante il predicato

```
hanoi(1,A,B,C).
```

Sposta il disco in cima alla pila dal piolo A al piolo C, usando il piolo B come appoggio.

Torri di Hanoi – caso base

Al fine di rendere visibili all'utente i movimenti effettuati per la risoluzione del problema, è possibile includere nel predicato alcune clausole di stampa, ottenendo il seguente predicato:

```
hanoi(1,A,B,C):-
        write('Muovi il disco in cima alla pila da'),
        write(A),
        write(' a '),
        write(C),
        nl.
```

Torri di Hanoi – caso ricorsivo

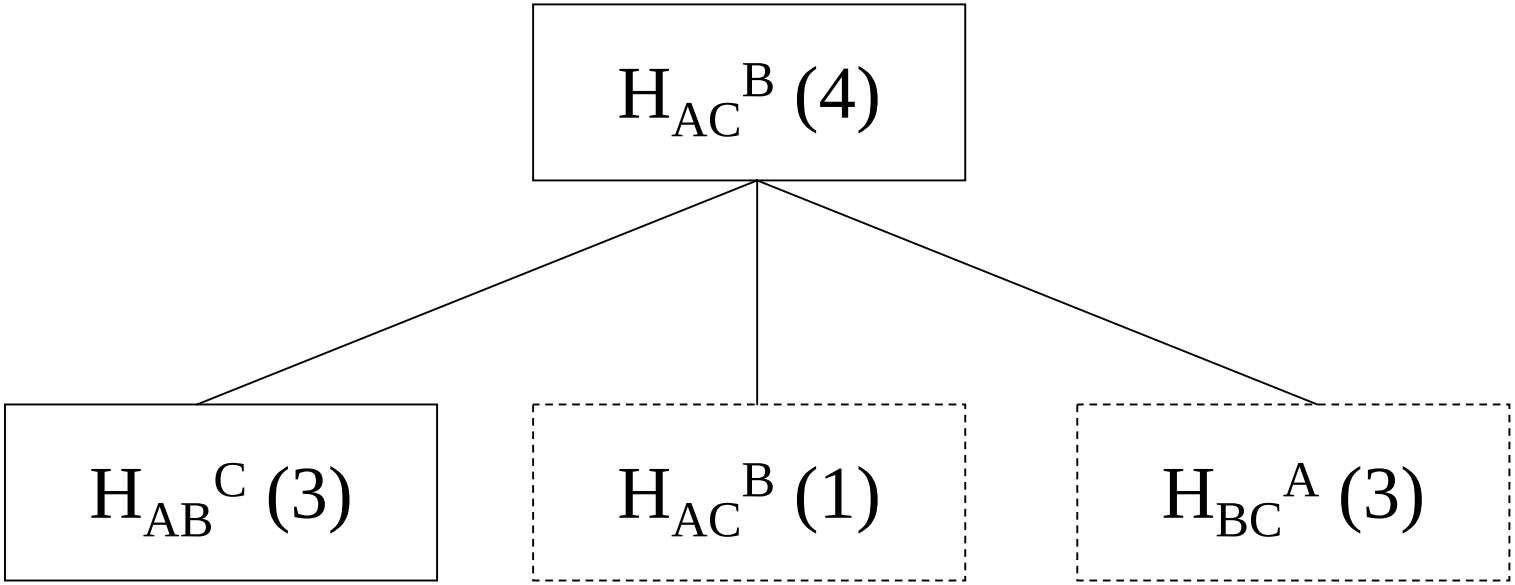
Se si hanno due o più pioli il gioco può essere risolto utilizzando la *riduzione a sottoproblemi*.

Si supponga ad esempio di avere ***n* = 4** dischi sulla pila di sinistra.

In questo caso si può procedere come segue:

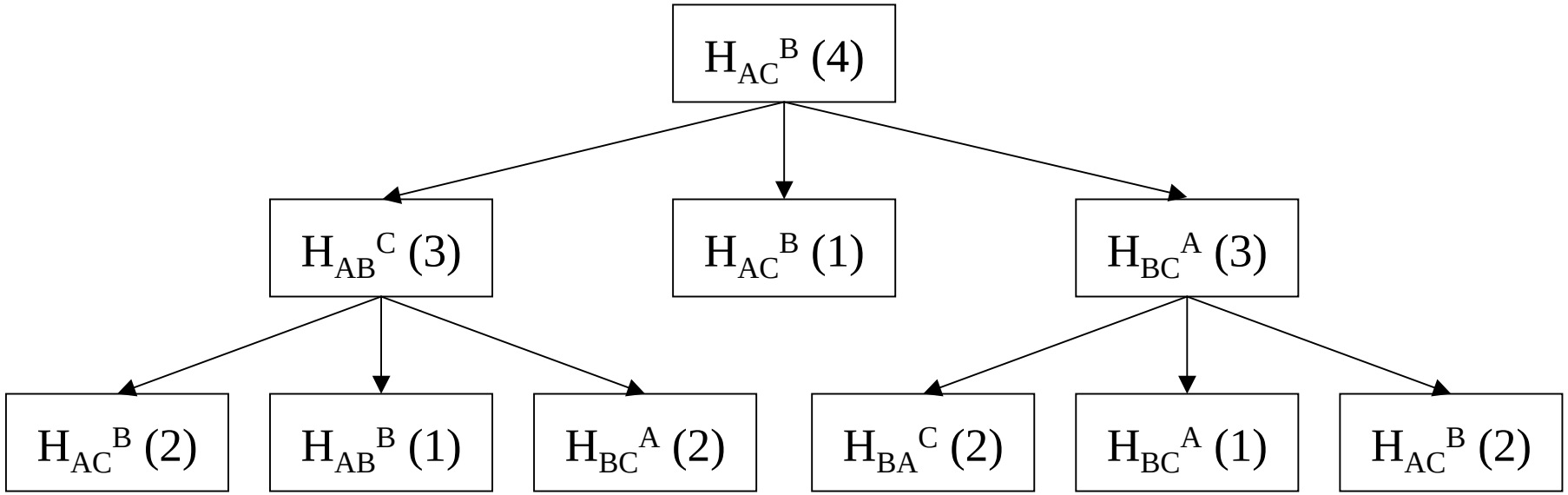
1. Si spostano i primi ***n-1*** dischi da sinistra al centro;
2. si sposta l'unico disco di sinistra sul piolo di destra;
3. si spostano gli ***n-1*** dischi dal piolo centrale a quello di destra.

Torri di Hanoi – caso ricorsivo



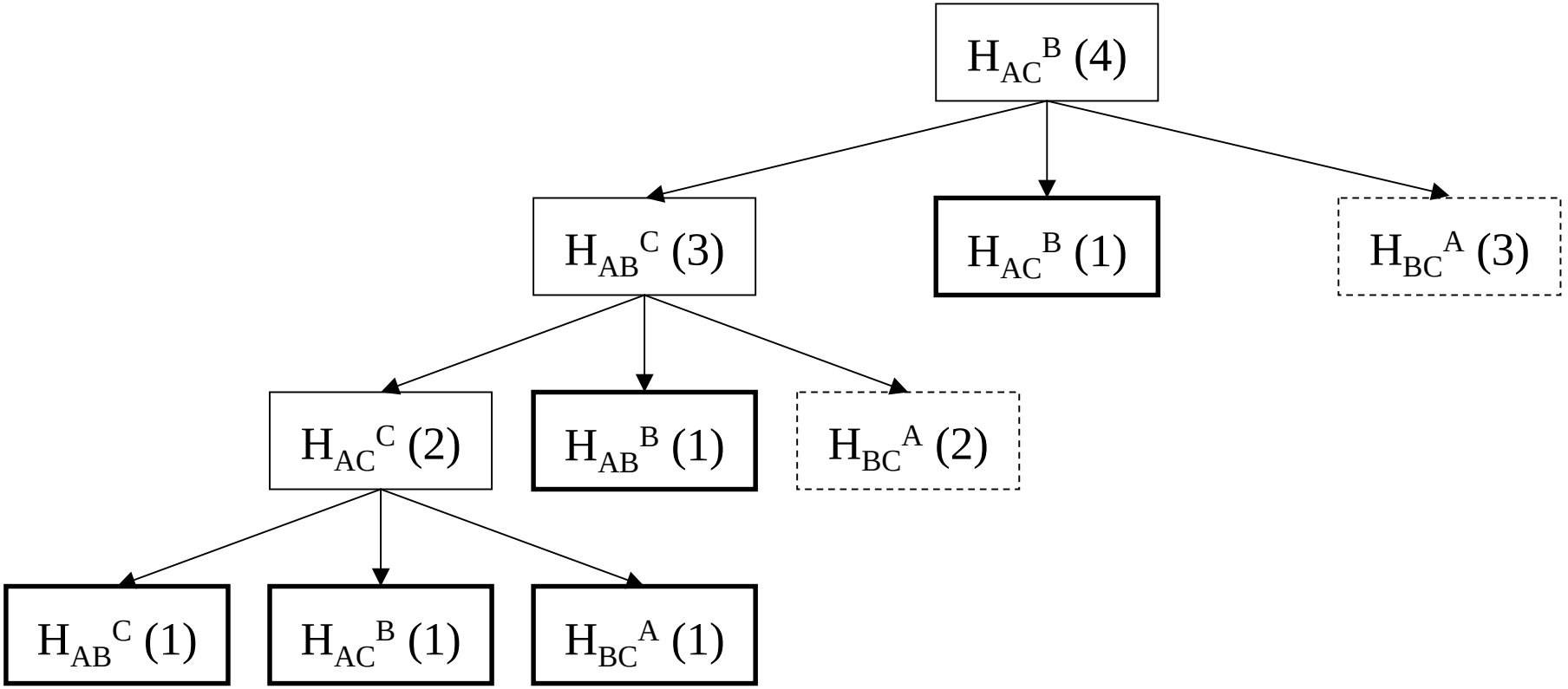
Torri di Hanoi – caso ricorsivo

Il problema è stato decomposto in tre problemi analoghi ma di dimensione minore. In particolare è stato decomposto in due problemi di dimensione ***n-1*** ed in un problema di dimensione **1**, il quale è banalmente risolto.



Torri di Hanoi – caso ricorsivo

Sviluppo parziale dell'albero di ricerca



Torri di Hanoi – caso ricorsivo

Il predicato Prolog che si può utilizzare per ricondurre un problema a ***n*** pioli ad un problema ad ***n-1*** pioli è il seguente:

```
hanoi(N,A,B,C):-
        N > 1,
        M is N-1,
        hanoi(M,A,C,B),
        hanoi(1,A,B,C),
        hanoi(M,B,A,C).
```

Torri di Hanoi – Programma completo

```
hanoi(1,Sinistra,Centro,Destra):-
    write('Muovi il disco in cima alla pila da'),
    write(Sinistra),
    write(' a '),
    write(Destra),
    nl.

hanoi(N,Sinistra,Centro,Destra):-
    N > 1,
    M is N-1,
    hanoi(M,Sinistra,Destra,Centro),
    hanoi(1,Sinistra,Centro,Destra),
    hanoi(M,Centro,Sinistra,Destra).
```

Torri di Hanoi - output

```
?- hanoi(3,sinistra,centro,destra).
```

Muovi il primo disco da sinistra a destra
Muovi il primo disco da sinistra a centro
Muovi il primo disco da destra a centro
Muovi il primo disco da sinistra a destra
Muovi il primo disco da centro a sinistra
Muovi il primo disco da centro a destra
Muovi il primo disco da sinistra a destra

Collegamenti

Si consideri il problema di rappresentare i collegamenti tra diverse città; inizialmente si considerino 4 città Roma, Na, Ba, RC.

Una possibilità è quella di realizzare un predicato coll con due argomenti, cioè due città tra loro collegate.

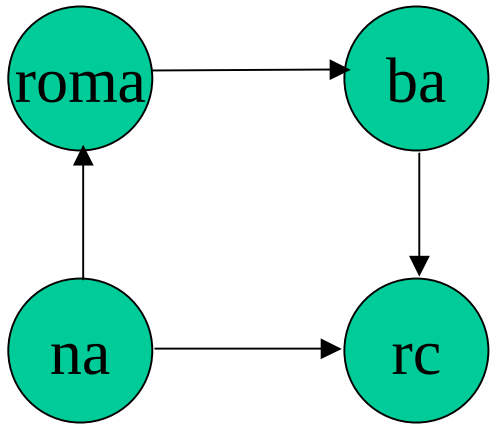
Esempio:

coll(roma,na) ⟷ Esiste un collegamento diretto tra Napoli e Roma

coll(na,rc).

coll(roma,ba).

coll(ba,rc).



Collegamenti

Il predicato definito è in grado di gestire soltanto il collegamento monodirezionale. Come gestire i collegamenti bidirezionali?
Una città è collegata con una seconda se la seconda è collegata con la prima $\exists coll(y,x) \Rightarrow \exists coll(x,y)$

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

Sottoponendo al sistema la query

```
?- coll(roma,na).
```

Il sistema non riesce a unificare con nessun assioma. Applicando la regola al goal, esso viene cambiato in `coll(na,roma)` il quale viene unificato.

Collegamenti

Sottoponendo al sistema la query

```
?- coll(na,X).
```

Il sistema troverà infinite unificazioni (Il MI costruisce un albero di ricerca con un ramo di lunghezza infinita e rimane ad esplorare quel ramo).

X = roma ;
X = rc ;
X = roma ;
X = rc ;
…

Collegamenti

Per eliminare la presenza del ramo di lunghezza infinita, bisogna costruire il predicato in modo diverso:

```
coll1(X,Y):-
        coll(Y,X).
coll1(X,Y):-
        coll(X,Y).
```

In questo modo l'utilizzo della ricorsione è stato eliminato.

Distanze

```
coll(na,roma,220).
coll(roma,fi,200).
coll(fi,bo,100).
coll(na,sa,60).
coll(sa,roma,280).
```

Se si sottopone al sistema la query `? coll(roma,na)`, il sistema non trova alcuna unificazione, in quanto la risposta alla query non è contenuta nella KB. Il sistema risulta quindi incompleto, se si assume che i collegamenti sono bidirezionali

Se si include l'informazione che i collegamenti sono bidirezionali, si può rendere il sistema completo rispetto ai collegamenti diretti presenti fra le 5 città in esame:

```
coll(X,Y,C) :-coll(Y,X,C).
```

Distanze

Il sistema è ora però diventato indecidibile, in quanto la regola aggiunta può creare dei loop. Provare ad esempio a effettuare la query

```
? coll(fi,ge,X).
```

Come nel caso dei collegamenti, per eliminare l'indecidibilità si può riformulare la KB come segue:

```
coll1(na,roma,220).
coll1(roma,fi,200).
coll1(fi,bo,100).
```

Distanze

```
coll1(na,sa,60).
coll1(sa,roma,280).
coll(X,Y,C) :-coll1(X,Y,C).
coll(X,Y,C) :-coll1(Y,X,C).
```

Esempio di query composta: quale città è collegata sia a Napoli che a Roma e quanto dista da entrambe?

```
?- coll(X,na,Y),coll(X,roma,Z).
```

```
X = sa
Y = 60
Z = 280;
```

Commesso Viaggiatore

Si vuole costruire un predicato per valutare quali siano i collegamenti presenti tra due qualsiasi città.
Si può partire dalla KB delle distanze ed inserire una regola. La KB a questo punto contiene una serie di città connesse direttamente e una regola per rendere il collegamento bidirezionale:

```
coll1(na,roma,220).
coll1(roma,fi,200).
coll1(fi,bo,100).
coll1(na,sa,60).
coll1(sa,roma,280).
coll(X,Y,C) :-coll1(X,Y,C).
coll(X,Y,C) :-coll1(Y,X,C).
```

Commesso Viaggiatore

È necessario definire:

una *città di partenza* **Ci**,

una *città di arrivo* **Cf**,

una *lista di città proibite* **Lp**,

un *cammino* **Lc** che congiunge **Ci** e **Cf** con *costo* **C**.

Sulla base di queste variabili, si può costruire un predicato che calcola tutti i percorsi **Lc** tra **Ci** e **Cf** con il costo **C**, evitando di passare per le città **Lp**.

Commesso Viaggiatore

La postcondizione può essere così strutturata:

```
collegamento(Ci,Cf,Lp,Lc,C):-...
```

Il caso base è

```
collegamento(Ci,Cf,Lp,[Ci,Cf],C):-
        coll(Ci,Cf,C),
        not(member(Ci,Lp)).
        not(member(Cf,Lp)).
```

Cioè esiste un collegamento diretto tra la città iniziale **Ci** e la città finale **Cf** ed entrambe non appartengono alla lista delle città proibite **Lp**.

Commesso Viaggiatore

A questo punto è possibile utilizzare la ricorsione per risolvere un problema di commesso viaggiatore analogo a quello proposto in partenza ma di dimensione minore, precisamente si vuole individuare un collegamento tra una città intermedia **Cint**, distante solo un passo dalla città iniziale **Ci** e la città finale **Cf**.

Nel far questo, bisogna impedire che la città **Ci** sia considerata nuovamente nel percorso, per evitare la possibilità di cicli. Si inserisce quindi **Ci** in testa alla lista delle città proibite.

Chiaramente la città iniziale **Ci** và inserita in testa alla lista **Lc** delle città inserite nel cammino.

Il predicato diviene quindi:

```
collegamento(Ci,Cf,Lp,[Ci|L1c],C):-
        coll(Ci,Cint,C1),
        collegamento(Cint,Cf,[Ci|Lp],L1c,C2).
```

Commesso Viaggiatore

Il significato è: Una città **Ci** è collegata ad una città **Cf**, senza passare per le città proibite **Lp**, seguendo il percorso **Lc=[Ci|L1c]**, e con costo **C**, se tale città **Ci** è collegata direttamente alla città intermedia **Cint** con costo **C1** e la città **Cint** è collegata alla città **Cf**, senza passare per le città proibite **[Ci|Lp]**, seguendo il percorso **L1c**, con costo **C2**.

```
collegamento(Ci,Cf,Lp,[Ci|L1c],C):-
        coll(Ci,Cint,C1),
        collegamento(Cint,Cf,[Ci|Lp],L1c,C2).
```

Commesso Viaggiatore

Il costo necessario a valutare il percorso da **Ci** a **Cf** passando per **Cint** è pari alla somma dei costi dei due percorsi, da **Ci** a **Cint** e da **Cint** a **Cf**, avendosi quindi

```
collegamento(Ci,Cf,Lp,[Ci|L1c],C):-
        coll(Ci,Cint,C1),
        collegamento(Cint,Cf,[Ci|Lp],L1c,C2),
        C is C1 + C2.
```

Commesso Viaggiatore

Ancora non è stato esplicitato il vincolo sulla lista delle città proibite: le città **Ci** e **Cint** possono appartenere al percorso soltanto se non sono membri della lista **Lp**.

```
collegamento(Ci,Cf,Lp,[Ci|L1c],C):-
        coll(Ci,Cint,C1),
        not(member(Ci,Lp)),
        not(member(Cint,Lp)),
        collegamento(Cint,Cf,[Ci|Lp],L1c,C2),
        C is C1 + C2.
```

Commesso Viaggiatore: il programma completo

```
collegamento(Ci,Cf,Lp,[Ci,Cf],C):-
        coll(Ci,Cf,C),
        not(member(Ci,Lp)).
        not(member(Cf,Lp)).

collegamento(Ci,Cf,Lp,[Ci|L1c],C):-
        coll(Ci,Cint,C1),
        not(member(Ci,Lp)),
        not(member(Cint,Lp)),
        collegamento(Cint,Cf,[Ci|Lp],L1c,C2),
        C is C1 + C2.
```

Predicati Extralogici
CUT, FAIL e NOT

Gli operatori CUT, FAIL e NOT sono predicati esterni alla logica del primo ordine e pertanto chiamati *operatori extralogici*. CUT, FAIL e NOT appartengono alla logica del secondo ordine, in quanto sono predicati il cui argomento è ancora un predicato.

L’operatore CUT

L’operatore ***cut*** è sintatticamente rappresentato dal simbolo “!”. Lo scopo del cut è quello di tagliare una parte dell’albero di ricerca.
Tagliando una parte dell’albero di ricerca è possibile che si alteri la completezza del sistema, in quanto se sono possibili unificazioni nella parte di albero di ricerca tagliato tramite il cut, tali unificazioni non vengono prese in considerazione.

Il cut serve per introdurre una forma di controllo sul meccanismo di backtracking automatico del motore inferenziale del Prolog.

L’operatore CUT

Si consideri il predicato A:- B_1, B_2 ,…, B_{i-1}, B_i,…, B_n.

Tale predicato è vero se i letterati B_1, B_2, B_3 fino a B_n hanno unificato. Se l’unificazione con B_i non riesce, viene applicato il *fail* su B_i ed il *redo* su B_{i-1} per tentare l’applicazione di un’altra regola presente nella KB (*backtracking*).

Volendo inserire il cut dopo la i-sima clausola, si ha:

A:- B_1, B_2 ,…, B_{i-1},!, B_i,…, B_n .

Il ruolo del cut è impedire il backtracking su B_i. Una volta applicato il *fail* su B_i, non può venire applicato il *redo*.

L’operatore CUT

La prima volta che il CUT viene incontrato (cioè scendendo lungo l’albero di ricerca) viene unificato. Se è stato incontrato il CUT la prima volta, vuol dire tutte le variabili presenti nelle clausole da B_1 a B_{i-1} sono state unificate. Se per unificare una qualsiasi delle variabili presenti nelle clausole da B_i a B_n bisogna liberare una delle variabili delle clausole da B_1 a B_{i-1} tutto il predicato fallisce, in quanto viene impedito il backtracking su quelle variabili.

Il predicato A risulterà quindi vero per una qualsiasi unificazione delle clausole da B_1 a B_{i-1} ma risulterà falso al primo tentativo di unificazione non riuscito su una delle variabili B_i,…, B_n.

Vengono quindi potenzialmente perse delle soluzioni, rendendo il sistema incompleto.

L’operatore CUT

Si consideri ad esempio il seguente predicato:

A:- B_1, B_2 ,…, B_{i-1},!,B_i,…, B_n.
A:- C_1, C_2 ,…, C_m.
A:- D_1, D_2 ,…, D_p.

Raggiunto il CUT, il backtracking è bloccato su *tutto* il predicato, anche sulle clausole in *or*. In molti casi quindi, le conseguenze dell’uso del CUT possono essere difficilmente prevedibili e l’uso è particolarmente delicato.

L’operatore CUT

Il cut può essere usato essenzialmente in due modi diversi.

CUT VERDE

Se è nota la posizione dell’unificazione nell’albero di ricerca, si può usare il cut per evitare esplorazioni successive inutili. In questo caso si parla di *cut verde*.

CUT ROSSO

Un altro uso del cut è quello di tagliare una parte dell’albero di ricerca in cui sono presenti delle soluzioni, in quanto si è interessati soltanto ad alcune di esse. In questo caso, il cut viene deliberatamente utilizzato per alterare la completezza del sistema, e si parlerà di *cut rosso*. Il cut rosso può essere facilmente fonte di effetti collaterali difficili da gestire.

Esempio di cut rosso

```
a(1).
a(2).

b(1).
b(2).

c(1).
c(2).
c(3).

d(1).
d(2).
d(3).

e(8).

p(X,Y) :- a(X),b(X),c(Y),d(Y).
p(X,Y) :- e(X),e(Y).
```

La query p(X,Y) fornisce 7 unificazioni.

?- p(X,Y).

X = 1 Y = 1 ;

X = 2 Y = 2 ;

X = 1 Y = 2 ;

X = 2 Y = 3 ;

X = 1 Y = 3 ;

X = 8 Y = 8 ;

X = 2 Y = 1 ;

Esempio di cut rosso

```
a(1).
a(2).

b(1).
b(2).

c(1).
c(2).
c(3).

d(1).
d(2).
d(3).

e(8).

p(X,Y) :- a(X),b(X),!,c(Y),d(Y).
p(X,Y) :- e(X),e(Y).
```

La query p(X,Y) fornisce
3 unificazioni.

?- p(X,Y).

X = 1
Y = 1 ;

X = 1
Y = 2 ;

X = 1
Y = 3 ;

Esempio di cut rosso

Nel programma logico in cui non è presente il cut, il motore inferenziale ha trovato sette unificazioni. L'inserimento del cut ha alterato il numero di soluzioni trovate.

Trovate le prime tre unificazioni va effettuato un passo di backtracking sulla clausola b(X). Tuttavia il CUT fa fallire il *redo* sulla clausola b(X), cioè impedisce che la variabile X, attualmente unificata, venga liberata in favore di una successiva unificazione, rendendo impossibili tutte le unificazioni che si sarebbero potute successivamente verificare.

E' da notare che durante la prima unificazione (*call*) il CUT viene ignorato.

Esempio di cut verde

Il programma logico proposto per l'eliminazione da una lista ha una prima unificazione che è quella che ci si aspetta. Tuttavia, a causa del fatto che le clausole sono poste in OR non esclusivo tra loro, vengono trovate altre unificazioni che non corrispondono ad eliminazioni.

Ciò che si vuole fare è quindi eliminare la possibilità di unificazioni successive alla prima. Per fare questo è possibile modificare il programma logico inserendo un CUT VERDE.

```
cancella(El,[],[]).
cancella(El,[El|C],C1) :- !,cancella(El,C,C1).
cancella(El,[T|C],[T|C1]) :- cancella(El,C,C1).
```

FAIL

Il FAIL è un predicato che si usa per inserire un controllo sul backtracking. Lo scopo del FAIL è complementare a quello del cut. Serve infatti a forzare un fallimento sull'unificazione in modo da consentire al backtracking di seguire un nuovo ramo.

Operatore NOT

```
not(P) :- P, !, fail.
not(P).
```

Se il predicato P è vero, la prima clausola risulta falsa a causa del *fail*, ed il *cut* fa fallire tutto il predicato.
Viceversa, se il predicato P è falso, la prima clausola è falsa, ma la seconda è vera (clausola di *catch-all*). È da notare che il predicato NOT funziona soltanto se l'ordine dei predicati in or è quello indicato e se l'ordine delle clausole all'interno del primo predicato è quello indicato. L'ordine dei predicati e delle clausole nei predicati diventa importante quando le clausole contengono operatori che non sono del primo ordine.

Operatore NOT

```
not(P) :- P, !, fail.
not(P).
```

Il significato del NOT è diverso dal *not* booleano:

Il *not* booleano ha successo sse il suo argomento è falso

Il NOT extralogico ha successo sse non è possibile alcuna unificazione delle variabili del predicato.

Esempio:

not(A(x_1, x_2, x_3, …, x_n))
è vero se non esiste alcuna unificazione contemporanea di tutte le varibili x_1, x_2, x_3, …, x_n