

Classe: tipo di dato definito dall'utente che rappresenta un insieme di oggetti con caratteristiche e comportamenti comuni.

è un componente che svolge il compito di assegnare un'etichetta a un'istanza di input.

6 Reti Neurali e Classificatori

l'insieme dei numeri interi tra 1 e m , inclusi

Per $n \in \mathbb{N}_+$, il **problema di classificazione** dei vettori di $S \subseteq \mathbb{R}^n$ è il problema che prevede di associare ogni vettore di S ad una **classe** scelta tra un numero finito e noto di classi tra loro disgiunte. Se $m \in \mathbb{N}_+$ è il numero di classi tra cui scegliere, un **classificatore** non è altro che una funzione f da S in $[1..m] \subset \mathbb{N}$. Siccome il problema di classificazione può essere ricondotto al problema di approssimare una funzione, è sicuramente possibile utilizzare un MLP come classificatore. Normalmente, si utilizza un MLP con n ingressi e m uscite addestrandolo in modo supervisionato mediante dei campioni che associano vettori di S a dei vettori del tipo

$$(0, 0, \dots, 1, \dots, 0) \quad (55)$$

dove il valore 1 viene messo nella posizione corrispondente alla classe del campione che si sta studiando. Quindi, ad esempio, se si sta usando un MLP per classificare vettori in 4 classi, allora se un campione è di classe 2 il relativo valore della funzione da approssimare viene fissato a $(0, 1, 0, 0)$.

Siccome l'addestramento della rete non riuscirà a garantire che i vettori calcolati abbiano solo componenti con valori pari a 0 o 1, è necessario che i vettori calcolati dalla rete vengano ulteriormente elaborati per stabilire quale sia la classe che la rete ha attribuito al vettore di input. Normalmente, si utilizza una tra due possibili metodi per stabilire la classe del vettore di input una volta noto il vettore di output. Il primo metodo prevede di scegliere in modo arbitrario un indice corrispondente ad una delle componenti massime del vettore calcolato dalla rete e, quindi, la classe stimata viene calcolata come segue

1° metodo
Tipo di classificazione
HARD MAX
↓
RIGIDO

Avendo un vettore di valori, esempio o_i (che sono le uscite della rete per il nodo $1, 2, \dots, m$), \max da solo dice quanto vale il massimo, $\arg \max$ dice la posizione del massimo

$$k = \operatorname{argmax}_{1 \leq i \leq m} o_i \quad (56)$$

vettore con m componenti

se il vettore di uscita della rete è $\mathbf{o} = (o_i)_{i=1}^m$. In alternativa, dato \mathbf{o} si può calcolare un nuovo vettore \mathbf{s} mediante la seguente funzione

Prendere il vettore di uscita \mathbf{o} e farlo passare attraverso la f.ne softmax , che ritorna un vettore che va da 1 a m , le cui componenti sono il risultato di questo calcolo.

$$\mathbf{s} = \operatorname{softmax}(\mathbf{o}) = \left(\frac{e^{o_i}}{\sum_{j=1}^m e^{o_j}} \right)_{i=1}^m$$

→ tutte le componenti tra 0 e 1, indipendentemente dagli o_i , che possono anche essere negative o grandi, e la somma di tutte le componenti di \mathbf{s} da 1, questo per come sono fatti gli esponenziali

2° metodo

Il vettore \mathbf{s} ha tutte le componenti comprese tra 0 e 1 e la somma di tutte le componenti risulta 1. Quindi, le componenti di \mathbf{s} possono essere interpretate come delle probabilità ed è possibile stimare la classe del vettore di ingresso estraendo un numero tra 1 e m ed assegnando ad ogni valore possibile $a \in [1..m]$ la probabilità s_a . In questo modo, classi a cui la rete ha associato valori alti sono comunque preferite pur non fissando un criterio rigido nella scelta tra classi con valori tra loro simili.

applico argmax su \mathbf{s}
 $k = \operatorname{argmax}_{1 \leq i \leq m} (s_i)$

Si noti che, com'è ragionevole attendersi, la capacità di un MLP di approssimare bene un classificatore dipende molto dal tipo di MLP che si sta considerando. Ad esempio, un $(n, 2)$ -SLP può essere usato per classificare vettori di \mathbb{R}^n in due classi facendolo seguire da un'elaborazione basata su argmax . In questo caso, la rete può solo comportarsi come un **classificatore lineare** e, quindi, la classificazione sarà totalmente corretta solo se esiste un iperpiano di \mathbb{R}^n in grado di separare i campioni appartenenti alle due classi. In alternativa, l'utilizzo di un MLP con almeno uno strato nascosto, oppure di una DNN, sarà in grado di produrre classificatori più raffinati e in grado di separare le due classi anche in casi complessi, come esemplificato in Figura 12. Si noti che la classificazione in due classi può essere affrontata anche mediante un MLP con un singolo neurone nello strato di output, come mostrato in Figura 12, senza comunque aumentare in alcun modo le capacità di classificazione della rete scelta.

*
Ricordo la definizione di IPERPIANO:

Dato il vettore reale $\mathbf{x} \in \mathbb{R}^n$ e il versore $\mathbf{n} \in \mathbb{R}^n$, l'iperpiano che contiene \mathbf{x} e ha \mathbf{n} come (versore) normale è

$$H_{\mathbf{n}}(\mathbf{x}) = \{ \mathbf{v} \in \mathbb{R}^n : (\mathbf{v} - \mathbf{x}) \cdot \mathbf{n} = 0 \} \quad (25)$$

↳ assicura che i vettori partano dalla fine di \mathbf{x}


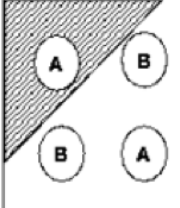
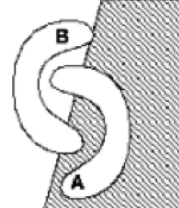

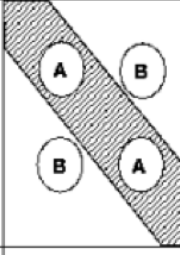
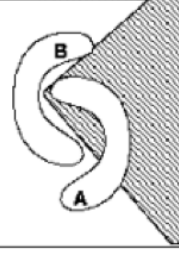

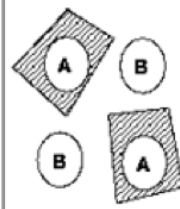
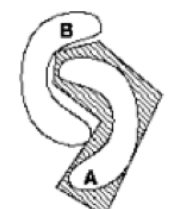
Structure	Regions	XOR	Meshed regions
single layer 	Half plane bounded by hyper-plane		
two layer 	Convex open or closed regions		
three layer 	Arbitrary (limited by # of nodes)		

Figura 12: Esempi di reti neurali usate come classificatori per 2 classi

Si noti che aumentare l'accuratezza con cui la rete classifica bene i vettori del training set ha spesso l'effetto di impedire alla rete di **generalizzare**. Quindi, ad esempio, nella Figura 12 l'utilizzo di due strati nascosti permette di migliorare le prestazioni della rete nel classificare i vettori della classe *A* ma rende la rete troppo permissiva nel classificare i vettori della classe *B*. Per questo, le prestazioni di un classificatore, indipendentemente dalla tecnica utilizzato per realizzarlo, vengono sempre espresse mediante la relativa **matrice di confusione** definita come $K = (k_{a,p})_{a,p=1}^m$, dove $k_{a,p}$ è la probabilità (condizionata) che il classificatore produca come stima della classe il valore p (da **predicted**) quando la classe effettiva è a (da **actual**). Naturalmente, un classificatore che non commette mai errori ha come matrice di confusione la matrice identità e , quindi, il comportamento di un classificatore è considerato tanto migliore quanto la sua matrice di confusione si avvicina alla matrice identità. Si noti, infine, che alle volte vengono introdotti degli **indici statistici** che riassumono le caratteristiche della matrice di confusione sotto opportune ipotesi.

7 Reti Neurali Ricorrenti e di Hopfield

Se una rete neurale non può essere descritta mediante una struttura a livelli, allora prende il nome di rete neurale **ricorrente** perché, almeno per un neurone, l'input dipende, eventualmente in modo indiretto, dal proprio output. Una rete neurale ricorrente è quindi un grafo orientato pesato i cui nodi sono neuroni di McCulloch-Pitts e, per ogni neurone, i pesi sugli archi sono i pesi associati al neurone. In generale, per una rete ricorrente non è più possibile identificare quale sia l'input e quale sia l'output, anche se è sempre possibile identificare quale sia l'input e quale sia l'output di un singolo neurone perché il grafo è

orientato. Anche in questo caso, per ogni neurone della rete, viene fissato uno dei valori di input ad un valore costante che viene detto valore di bias e normalmente vale -1 . Se non viene esplicitamente indicato il contrario, una rete neurale ricorrente si intende descritta da un grafo completamente connesso privo di autoanelli.

Una rete (neurale) di Hopfield è una rete neurale ricorrente in cui la funzione di attivazione dei neuroni è la funzione **signum**

$$\text{sgn}(x) = \begin{cases} +1 & \text{se } x > 0 \\ 0 & \text{se } x = 0 \\ -1 & \text{se } x < 0 \end{cases} \quad (58)$$

Visto che non è stato detto altrimenti, una rete di Hopfield è descritta da un grafo completamente connesso privo di autoanelli. \rightarrow (non esiste un arco che collega un neurone direttamente con se stesso)

L'ingresso della rete viene fornito fissando i valori di uscita dei neuroni a valori iniziali nell'insieme $\{-1, +1\}$. L'uscita della rete viene ottenuta leggendo i valori delle uscite dei neuroni dopo un tempo di elaborazione sufficientemente lungo, nell'ipotesi che le uscite si siano stabilizzate a valori asintotici, se esistono. Si noti che, di norma, il valore 0 non viene usato come ingresso e si ipotizza che venga prodotto come uscita solo durante le fasi transitorie in cui l'uscita non è ancora considerata stabile. Quindi, una rete di Hopfield legge un ingresso binario e produce un'uscita binaria, sempre nell'insieme $\{-1, +1\}$.

Una rete di Hopfield è quindi un **sistema dinamico non lineare** il cui comportamento può essere studiato in modo analitico sotto opportune ipotesi. Normalmente, però, una rete di Hopfield viene **simulata a tempo discreto** aggiornando l'uscita di un neurone scelto casualmente per ogni istante di simulazione. Quindi, data una rete di Hopfield, si fissano inizialmente i valori delle uscite dei neuroni e poi, ciclicamente, si sceglie un neurone casualmente e si aggiorna la sua uscita sulla base dei relativi ingressi, fino al raggiungimento di un numero sufficiente di iterazioni o fino alla stabilizzazione dei valori di uscita dei neuroni.

Data una rete di Hopfield con $n \in \mathbb{N}_+$ neuroni, fissato un qualsiasi istante di simulazione, il vettore $\mathbf{s} \in \{-1, 0, +1\}^n$ formato delle uscite dei neuroni prima dell'aggiornamento delle uscite stesse viene detto **stato della rete**. Lo stato della rete evolve dinamicamente partendo da uno stato iniziale corrisponde al vettore di ingresso della rete e giungendo ad uno stato finale corrisponde al vettore di uscita della rete. Questo tipo di simulazione del comportamento di una rete di Hopfield viene detto **simulazione asincrona** e, contrariamente alla **simulazione sincrona** che non verrà discussa qui, descrive bene la rete nei termini di un sistema dinamico.

Si consideri una rete di Hopfield formata da $n \in \mathbb{N}_+$ neuroni, ognuno dei quali è associato ad un vettore dei pesi e ad un valore di bias che vengono entrambi usati per produrre un valore di uscita. I neuroni della rete vengono normalmente numerati da 1 a n e, quindi, l' i -esimo neurone, con $1 \leq i \leq n$, viene descritto da un vettore dei pesi $\mathbf{w}_i \in \mathbb{R}^n$ e da un valore di bias $b_i \in \mathbb{R}$, dove si assume che $w_{i,i} = 0$ perché nella rete non sono presenti autoanelli. Se $\mathbf{s} \in \mathbb{R}^n$ è lo stato attuale della rete, allora l'uscita dell' i -esimo neurone vale

$$o_i = \text{sgn}(\mathbf{w}_i \cdot \mathbf{s} - b_i) = \text{sgn} \left(\sum_{j=1}^n \underbrace{w_{i,j}}_{\text{pesi}} \underbrace{s_j}_{\text{stato attuale di rete}} - \underbrace{b_i}_{\text{peso bias}} \right) \quad (59)$$

f.ne attiv. signum

Si noti che, per $1 \leq i \leq n$, $o_i \in \{-1, 0, +1\}$. In più, si noti che l'uscita dell' i -esimo neurone si dice **stabile**, o **stazionaria**, se o_i non varia anche dopo l'aggiornamento causato da una variazione dello stato della rete.

\rightarrow l'output del neurone (o_i) è uguale all'input (s). Se succede che in una rete, da un certo momento, tutti gli o_i sono uguali agli s_i , allora la rete è stabile.

*
Matrice simmetrica: matrice quadrata in cui ogni elemento della diag. principale è uguale al suo simmetrico rispetto alla diagonale principale.

Se A è una matrice $n \times n$, allora

$$A[i,j] = A[j,i]$$

Ricordo che

$$A \times A^T = A$$

↑
trasposta: matrice con righe e colonne invertite

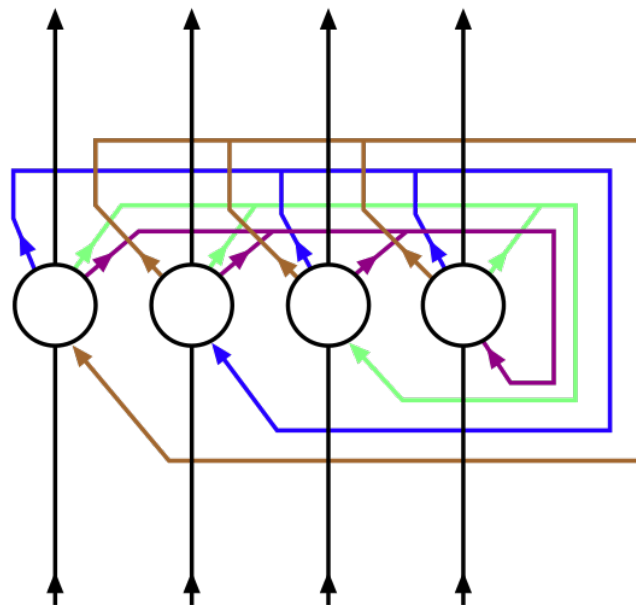


Figura 13: Esempio di reti di Hopfield con 4 neuroni

Si consideri una rete di Hopfield formata da $n \in \mathbb{N}_+$ neuroni, ognuno dei quali è associato ad un vettore dei pesi $\mathbf{w}_i \in \mathbb{R}^n$, con $1 \leq i \leq n$, è possibile definire la **matrice dei pesi della rete** $W = (w_{i,j})_{n,n}$. Si noti che, come già discusso, la matrice dei pesi ha solo zeri sulla diagonale principale perché la rete non presenta autoanelli. Fissata una rete di Hopfield mediante la relativa matrice dei pesi della rete W e il relativo **vettore di bias** \mathbf{b} , è possibile definire l'**energia della rete** associata ad uno stato $\mathbf{s} \in \mathbb{R}^n$ della rete come

$$E(\mathbf{s}) = -\frac{1}{2} \mathbf{s}^T W \mathbf{s} + \mathbf{b} \cdot \mathbf{s} = -\frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n w_{i,j} s_i s_j + \sum_{i=1}^n b_i s_i \quad (60)$$

serve per indicare che decresce (pointing to the negative sign)
matrice pesi (pointing to W)
vettore di n componenti con solo -1 o 1 (pointing to s)
vettore bias (pointing to b)

Si noti che, come di consueto, i vettori di \mathbb{R}^n vengono trattati come vettori colonna quando vengono considerati insieme a delle matrici. In più, si noti che la parola **energia** è stata scelta per la quantità $E(\mathbf{s})$ perché vale il seguente teorema.

Teorema 2 (Di Convergenza delle Reti Neurali di Hopfield). *Si consideri una rete neurale di Hopfield formata da $n \in \mathbb{N}_+$ neuroni e descritta da una matrice dei pesi W e da un vettore di bias \mathbf{b} . Se la matrice dei pesi è simmetrica*, allora, indipendentemente dallo stato iniziale, tutti i neuroni della rete tenderanno ad uno stato stazionario che corrisponde ad un minimo locale dell'energia della rete.*

Dimostrazione. Si consideri un passo della simulazione asincrona della rete e sia $1 \leq a \leq n$ l'indice del neurone il cui stato viene aggiornato nel passo considerato. Se $\mathbf{s} \in \mathbb{R}^n$ è lo stato della rete prima del passo di simulazione e $\mathbf{s}' \in \mathbb{R}^n$ è lo stato della rete dopo il passo di simulazione, allora la variazione dell'energia da $E(\mathbf{s})$ a $E(\mathbf{s}')$ vale $\Delta E = E(\mathbf{s}') - E(\mathbf{s})$

$$\Delta E = -\frac{1}{2} \left(\sum_{i=1}^n w_{i,a} s'_i s'_a + \sum_{j=1}^n w_{a,j} s'_a s'_j - \sum_{i=1}^n w_{i,a} s_i s_a - \sum_{j=1}^n w_{a,j} s_a s_j \right) + b_a (s'_a - s_a) \quad (61)$$

dove si è sfruttato il fatto che $w_{a,a} = 0$ per ipotesi. Ma, siccome, per ipotesi, la matrice dei pesi della rete è simmetrica, vale

$$\Delta E = - \left(\sum_{i=1}^n w_{i,a} s'_i s'_a - \sum_{i=1}^n w_{i,a} s_i s_a \right) + b_a (s'_a - s_a) \quad (62)$$

e quindi

$$\begin{aligned} \Delta E &= - \left(s'_a \sum_{i=1}^n w_{i,a} s'_i - s_a \sum_{i=1}^n w_{i,a} s_i \right) + b_a (s'_a - s_a) \\ &= -(s'_a - s_a) \left(\sum_{i=1}^n w_{i,a} s_i - b_a \right) \end{aligned} \quad (63)$$

perché, nel passo di aggiornamento considerato, solo s_a può cambiare e quindi, per ogni $1 \leq i \leq n$, con $i \neq a$, vale $s'_i = s_i$. In più, si noti che

$$\left(\sum_{i=1}^n w_{i,a} s_i - b_a \right) \quad (64)$$

↗ Preattivazione

è il valore dell'uscita del neurone a prima dell'applicazione della funzione signum e quindi il suo segno è uguale al segno di s'_a . Quindi,

- Se l'uscita del neurone a non varia a causa dell'aggiornamento, allora $\Delta E = 0$ e $E'(\mathbf{s}) = E(\mathbf{s})$;
- Se l'uscita del neurone a varia a causa dell'aggiornamento e diventa 0, allora $\Delta E = 0$ e $E'(\mathbf{s}) = E(\mathbf{s})$;
- Se l'uscita del neurone a varia a causa dell'aggiornamento e diventa +1, allora $\Delta E < 0$ e $E'(\mathbf{s}) \leq E(\mathbf{s})$;
- Se l'uscita del neurone a varia a causa dell'aggiornamento e diventa -1, allora $\Delta E < 0$ e $E'(\mathbf{s}) \leq E(\mathbf{s})$.

Quindi, in sintesi, l'evoluzione dinamica dello stato della rete può solo fare decrescere o rimanere inalterata l'energia della rete, che quindi arriverà ad assestarsi asintoticamente ad un minimo locale. \square

Si noti che, fissata una rete di Hopfield, E non è l'unica funzione che non cresce mai durante l'evoluzione dinamica della rete e, quindi, esistono altre funzioni che possono essere usate come energia della rete.

Una rete di Hopfield può essere usata come **memoria associativa** se, dato un insieme di vettori di ingresso detto training set, è possibile costruire una rete il cui stato evolva dinamicamente e si assesti ad uno dei vettori del training set **simile** al vettore di ingresso. In sostanza, quindi, lo stato della rete viene inizializzato con un vettore di ingresso e si prevede che la rete converga asintoticamente ad uno stato, tra quelli nel training set, per cui è alta la somiglianza con lo stato di ingresso.

Dato un training set, l'addestramento di una rete di Hopfield prevede di calcolare una matrice dei pesi e un vettore di bias che si comportino bene quando utilizzati con un adeguato test set. Si noti che, come si evince dal seguente teorema, l'addestramento di una rete di Hopfield è **non supervisionato** perché non si prevede che i vettori del training

* MUTUALMENTE ORTOGONALI

Un insieme finito di vettori reali non nulli $O \subset \mathbb{R}^n$ si dice formato da vettori **mutualmente ortogonali** se e soltanto se, qualsiasi siano $\mathbf{x} \in O$ e $\mathbf{v} \in O$, vale $\mathbf{x} \cdot \mathbf{v} = 0$. Si noti che gli insiemi di vettori mutuamente ortogonali sono particolarmente interessanti perché vale la seguente proposizione.

Proposizione 3. Dato $n \in \mathbb{N}_+$, se $O \subset \mathbb{R}^n$ è un insieme di vettori reali mutuamente ortogonali, allora è anche un insieme di vettori linearmente indipendenti. *

set siano associati a valori di uscita corretti. Questo rende le reti di Hopfield più semplici da utilizzare con training set di grandi dimensioni perché non è necessario associare ad ogni vettore del training set il corrispettivo valore atteso.

Teorema 3 (Di Addestramento con la Regola di Hebb). *nota il training set, ne calcola la matrice dei pesi e garantisce che essa, sia, proprio la matrice cercata, cioè che abbia i minimi locali nei punti del training set*
Data una rete neurale di Hopfield con $n \in \mathbb{N}_+$ neuroni, si consideri un training set $T = \{\mathbf{x}_k\}_{k=1}^m$ formato da $m \in \mathbb{N}_+$ vettori di \mathbb{R}^n tra loro mutuamente ortogonali* e privi di componenti nulle, se la rete ha vettore di bias nullo e matrice dei pesi

$$W = \frac{1}{m} \sum_{k=1}^m \mathbf{x}_k \mathbf{x}_k^T - I_n \quad (65)$$

dove I_n è la matrice identità $n \times n$, allora l'energia della rete ammette minimi locali in corrispondenza dei vettori del training set.

Dimostrazione. Per prima cosa si noti che i vettori del training set sono vettori di \mathbb{R}^n e, quindi, sarà $m \leq n$ perché è noto che un insieme di vettori mutuamente ortogonali è anche un insieme di vettori linearmente indipendenti. In più, si noti che la matrice considerata ha i seguenti elementi, per $1 \leq i \leq n$ e $1 \leq j \leq n$,

$$w_{i,j} = \frac{1}{m} \sum_{k=1}^m x_{k,i} x_{k,j} - \delta_{i,j} \quad (66)$$

dove $\delta_{i,j}$ è la **delta di Kronecker** (visto in CALCOLO NUMERICO)

$$\delta_{i,j} = \begin{cases} 1 & \text{se } i = j \\ 0 & \text{se } i \neq j \end{cases} \quad (67)$$

Quindi, la matrice W è simmetrica e ha solo zeri sulla diagonale principale perché si è ipotizzato che i vettori del training set non abbiano componenti nulle. Si consideri la derivata dell'energia rispetto ad una generica componente dello stato della rete di indice $1 \leq a \leq n$

$$\frac{\partial E}{\partial s_a}(\mathbf{s}) = -\frac{1}{2} \left(\sum_{i=1}^n w_{i,a} s_i + \sum_{j=1}^n w_{a,j} s_j \right) \quad (68)$$

dove si è sfruttato il fatto che $w_{a,a} = 0$ e si è ricordato che, per ipotesi, il vettore di bias della rete è nullo. Siccome la matrice dei pesi considerata è simmetrica, vale

$$\frac{\partial E}{\partial s_a}(\mathbf{s}) = -\sum_{i=1}^n w_{i,a} s_i \quad (69)$$

e quindi

$$\frac{\partial E}{\partial s_a}(\mathbf{s}) = -\frac{1}{m} \sum_{k=1}^m \sum_{i=1}^n s_i x_{k,i} x_{k,a} + \sum_{i=1}^n \delta_{i,a} s_i \quad (70)$$

Si ricordi ora che i vettori del training set sono tra loro mutuamente ortogonali e quindi, per ogni $1 \leq k \leq m$ e $1 \leq h \leq m$, con $h \neq k$, vale

$$\mathbf{x}_k \cdot \mathbf{x}_h = \sum_{r=1}^n x_{k,r} x_{h,r} = 0 \quad (71)$$

Quindi, se la derivata dell'energia viene valutata in un generico vettore del training set di indice $1 \leq h \leq m$, vale

derivata dell'Energia in uno stato a in un vettore del training set h $\leftarrow \frac{\partial E}{\partial s_a}(\mathbf{x}_h) = -\frac{1}{m} \sum_{i=1}^n m x_{h,a} + x_{h,a} = 0 \quad (72)$

perché, siccome i vettori del training set non hanno componenti nulle, vale che $x_{h,i} x_{h,i} = 1$ per ogni $1 \leq h \leq m$ e $1 \leq i \leq n$. \square

Si noti che il teorema precedente non assicura che gli unici minimi locali della funzione energia siano quelli relativi ai vettori del training set e, quindi, fissato uno stato iniziale, non è garantito che la rete converga proprio ad uno dei vettori del training set. In generale, quindi, il **richiamo** di un vettore dalla rete, cioè la sua lettura a fronte di un vettore di ingresso, potrebbe essere non esatto. In sintesi, nonostante l'addestramento di una rete di Hopfield non sia supervisionato, l'accuratezza nel richiamo dei vettori deve sempre essere quantificata mediante un'analisi con un opportuno test set.

Seguono alcuni esempi di richiamo di vettori mediante una rete addestrata con la **regola di Hebb**, che è la regola per la costruzione della matrice dei pesi del teorema precedente. Si noti che i vettori utilizzati per l'addestramento non sono mutuamente ortogonali, ma viene comunque usata la regola di Hebb ipotizzando di ottenere comunque buoni risultati.

Si consideri una rete di Hopfield in cui è stato memorizzato, mediante la regola di Hebb, il vettore mostrato nella parte destra della Figura 14.

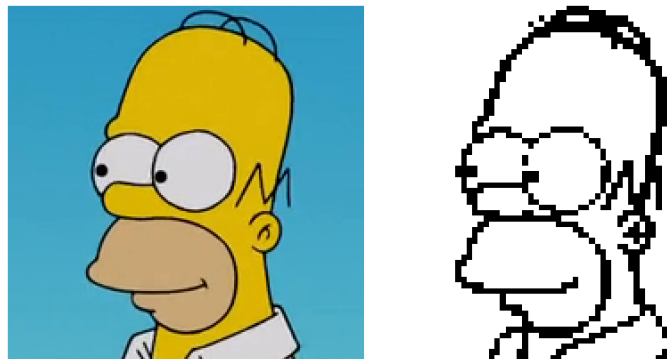


Figura 14: Immagine usata per alcuni esperimenti con una rete di Hopfield con $n = 64 \times 64$ neuroni (sinistra) e il relativo vettore binario usato per l'addestramento della rete (destra)

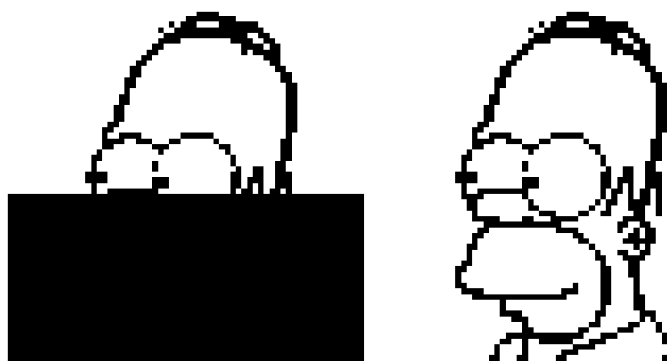


Figura 15: Vettore usato come ingresso della rete di Hopfield addestrata con il vettore di Figura 14 (sinistra) e l'uscita ottenuta dalla rete (destra)

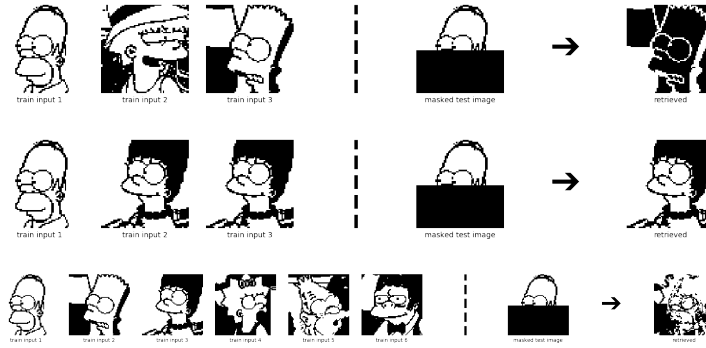


Figura 16: Alcuni esempi in cui la regola di Hebb per l'addestramento di una rete di Hopfield non consente di richiamare correttamente alcuni vettori

A fronte del vettore mostrato in Figura 15 la rete ha richiamato il vettore originale, come richiesto. Quindi, in questo caso, la rete è stata in grado di richiamare il vettore corretto anche in presenza di un forte rumore.

Però, la stessa rete, una volta che è stato aumentato il numero di vettori memorizzati, come mostrato in Figura 16 inizia a comportarsi male e il suo comportamento peggiora al crescere del numero di vettori memorizzati. Si noti che gli errori nel richiamo dei vettori non sono necessariamente legati all'eccessivo numero di vettori memorizzati nella rete. Ad esempio, in questo caso, il numero di vettori memorizzati è sufficientemente piccolo da ritenere che gli errori non siano dovuti alla quantità di vettori memorizzati, ma alla loro dipendenza lineare. In questo caso, infatti, si vede come la rete tenda a richiamare dei vettori ottenuti componendo vari vettori memorizzati.

Appunti del Corso di Intelligenza Artificiale

Problemi di Soddisfacimento di Vincoli

Prof. Federico Bergenti

21 aprile 2024

1 Problemi di Soddisfacimento di Vincoli

Un **problema di soddisfacimento di vincoli** (**Constraint Satisfaction Problem, CSP**) è una tripla $\langle V, D, C \rangle$ in cui

- $V \neq \emptyset$ è un insieme non vuoto e finito di simboli detti variabili con $n = |V|$;
- $D \neq \emptyset$ è un insieme non vuoto di insiemi detti **domini** delle variabili con $|D| \leq n$; e
- C è un insieme finito di **vincoli**.

Detta $dom : V \rightarrow D$ una funzione suriettiva totale che associa un dominio ad ogni variabile, si può parlare, per ogni variabile $x \in V$, del suo dominio $dom(x)$. In più, si suppone sempre che esista un ordinamento totale delle variabili e che questo ordinamento venga sempre usato in senso crescente quando vengono elencate le variabili. Normalmente, l'ordinamento utilizzato per le variabili è lasciato implicito ed evidente dal contesto. Utilizzando l'ordinamento delle variabili è possibile definire il dominio di un CSP \mathcal{P} come

$$dom(\mathcal{P}) = \prod_{x \in V} dom(x) \quad (1)$$

dove V è l'insieme delle variabili del CSP e le variabili vengono elencate nella costruzione del prodotto cartesiano in senso crescente secondo l'ordinamento scelto.

Dato un CSP con variabili in V , ogni vincolo $c \in C$ è una coppia $\langle V_c, \Delta_c \rangle$ dove $V_c \subseteq V$ è un insieme di variabili del CSP e Δ_c è

$$\Delta_c \subseteq \prod_{x \in V_c} dom(x) \quad (2)$$

dove le variabili vengono elencate nella costruzione del prodotto cartesiano in senso crescente secondo l'ordinamento scelto e Δ_c viene detto insieme degli **assegnamenti (parziali) consistenti (o ammissibili)** del vincolo c . Se $|V_c| = 1$, allora c viene detto **unario**, se $|V_c| = 2$, allora c viene detto **binario**, mentre c viene detto **globale** in tutti gli altri casi. Infine, si noti che $vars(c) = V_c$ è un modo per fare riferimento alle variabili di un vincolo e

$$dom(c) = \prod_{x \in V_c} dom(x) \quad (3)$$

viene detto dominio del vincolo c se si assume che le variabili vengano elencate in senso crescente secondo l'ordinamento scelto.

Fissato un CSP \mathcal{P} , è sempre possibile estendere Δ_c ad un sottoinsieme del dominio del CSP $\Delta = \text{dom}(\mathcal{P})$ aggiungendo alle ennuple di Δ_c le componenti mancanti. Siccome se una variabile non è coinvolta in un vincolo i suoi valori sono tutti e soli quelli del proprio dominio, le componenti aggiunte alle ennuple di Δ_c potranno assumere un valore qualsiasi nei domini delle rispettive variabili. Detta *img* una funzione suriettiva totale che associa ad ogni vincolo l'estensione a Δ dell'insieme degli assegnamenti (parziali) consistenti, si può parlare, per ogni vincolo $c \in C$, del suo insieme degli **assegnamenti totali consistenti** (o **ammissibili**) $\text{img}(c) \subseteq \Delta$.

Se tutti i domini delle variabili di un vincolo sono finiti, allora è possibile descrivere il vincolo elencando tutti gli assegnamenti parziali consistenti in modo **estensionale** e il vincolo viene detto **tabellare**. La rappresentazione tabellare diventa impraticabile anche per domini con pochi elementi e quindi spesso si preferisce la rappresentazione **intensionale**.

Esempio 1. Si consideri un CSP con tre variabili, x , y e z , tali che $\text{dom}(x) = \text{dom}(y) = \text{dom}(z) = [1..6]$. L'insieme delle variabili del CSP è $V = \{x, y, z\}$ e il dominio del CSP è $\Delta = [1..6]^3$ dove, come di consueto, le variabili vengono ordinate in senso alfabetico. Il CSP contiene un vincolo c definito dalla proprietà

$$x \text{ e } z \text{ devono essere diversi ed entrambi pari}$$

Quindi, $c = \langle V_c, \Delta_c \rangle$ dove $V_c = \{x, z\}$ e Δ_c può essere espresso in forma estensionale come segue

$$\Delta_c = \{(2, 4), (2, 6), (4, 2), (4, 6), (6, 2), (6, 4)\} \quad (4)$$

dove, come di consueto, è stato utilizzato l'ordinamento alfabetico per le variabili. Si noti che l'estensione di Δ_c ad un sottoinsieme di Δ può essere espressa in forma estensionale come segue

$$\text{img}(c) = \{(2, 1, 4), (2, 2, 4), \dots, (6, 6, 4)\} \quad (5)$$

dove ogni elemento di Δ_c è stato utilizzato per produrre 6 elementi di $\text{img}(c)$. L'insieme degli assegnamenti parziali consistenti di c può essere espresso in forma intensionale come segue

$$\Delta_c = \{(x, z) : x \in I \wedge z \in I \wedge x \neq z \wedge x \bmod 2 = 0 \wedge z \bmod 2 = 0\} \quad (6)$$

e quindi

$$\text{img}(c) = \{(x, y, z) : x \in I \wedge y \in I \wedge z \in I \wedge x \neq z \wedge x \bmod 2 = 0 \wedge z \bmod 2 = 0\} \quad (7)$$

La possibilità di estendere gli insiemi degli assegnamenti parziali consistenti ad insiemi di assegnamenti totali consistenti permette di definire in modo semplice cosa si intenda per soluzione di un CSP. Dato un CSP $\mathcal{P} = \langle V, D, C \rangle$, l'insieme delle **soluzioni** di \mathcal{P} è

$$\text{sol}(\mathcal{P}) = \bigcap_{c \in C} \text{img}(c) \quad (8)$$

e il \mathcal{P} si dice **risolubile** (o **risolvibile**) se questo insieme non è vuoto. Dati due CSP \mathcal{P}_1 e \mathcal{P}_2 definiti sullo stesso insieme di variabili, si dice che \mathcal{P}_1 è **equivalente** a \mathcal{P}_2 se $\text{sol}(\mathcal{P}_1) = \text{sol}(\mathcal{P}_2)$. Si noti che questa definizione può essere estesa a problemi con lo stesso numero di variabili rinominando opportunamente le variabili dei problemi.

Dato un CSP è interessante cercare delle trasformazioni in grado di generare dei CSP ad esso equivalenti ma che siano più utili per la ricerca delle soluzioni. In generale, però,

sono interessanti anche le trasformazioni in grado di produrre dei CSP non equivalenti a quello di partenza ma che permettano, comunque, di studiare le proprietà di quest'ultimo.

Esempio 2. Si consideri un CSP \mathcal{P} con due variabili x e y entrambe con dominio $[-10..10]$. Il CSP ha un unico vincolo descritto dalla proprietà $y = x^2$. Si noti che il CSP è risolubile perché, ad esempio, $x = -2$ e $y = 4$ è una soluzione. Il CSP \mathcal{P}_1 che restringe il dominio di y a $[0..10]$ è equivalente a \mathcal{P} perché è facile vedere che entrambi ammettono lo stesso insieme di soluzioni. Viceversa, il CSP \mathcal{P}_2 che restringe anche il dominio di x all'insieme $[0..10]$ non è equivalente a \mathcal{P} perché, ad esempio, $x = -2$ e $y = 4$ è una soluzione di \mathcal{P} ma non è una soluzione di \mathcal{P}_2 . Però, il CSP \mathcal{P}_2 , che è ottenuto da \mathcal{P} con una trasformazione detta **di rottura della simmetria** (o **symmetry breaking**), è comunque interessante perché una volta risolto è possibile ottenere le soluzioni di \mathcal{P} . In più, \mathcal{P}_2 ha un numero inferiore di elementi nel dominio di x e quindi la ricerca esaustiva delle soluzioni è facilitata.

Dato un CSP, ogni trasformazione che produce un CSP ad esso equivalente ma con meno vincoli o meno valori nei domini viene detta **propagazione dei vincoli**. Se la trasformazione utilizza un sottoinsieme dei vincoli, allora viene detta **propagazione locale (dei vincoli)**. Se la trasformazione utilizza un sottoinsieme dei vincoli per rimuovere alcuni valori dai domini delle variabili coinvolte nei vincoli considerati, allora viene detta **filtro (dei valori)**.

Normalmente, le trasformazioni tra CSP equivalenti vengono utilizzate per ottenere nuovi CSP con proprietà interessanti. Ad esempio, una trasformazione che spesso viene utilizzata permette di ottenere un CSP nodo-consistente equivalente ad un CSP dato. Si noti che un CSP si dice **nodo-consistente** (o **node-consistent**) se per ogni vincolo unario c vale la seguente proprietà

$$\forall v \in \text{dom}(x), v \in \Delta_c \quad (9)$$

dove x è l'unica variabile del vincolo c e Δ_c è l'insieme degli assegnamenti parziali consistenti di c .

Un **algoritmo di nodo-consistenza** lavora su un CSP per produrre un CSP nodo-consistente ad esso equivalente eliminando dai domini delle variabili tutti i valori che non rispettano la definizione precedente. In più, un algoritmo di nodo-consistenza rimuove anche tutti i vincoli unari dal CSP perché, una volta eliminati i valori inconsistenti dai domini, i vincoli unari non sono più necessari.

In modo simile, un **algoritmo di arco-consistenza** lavora su un CSP per produrre un CSP arco-consistente ad esso equivalente eliminando dai domini delle variabili tutti i valori che non rispettano la definizione di arco-consistenza. Un CSP si dice **arco-consistente** (o **arc-consistent**) se per ogni vincolo binario c valgono le seguenti proprietà

$$\forall v_x \in \text{dom}(x), \exists v_y \in \text{dom}(y) : (v_x, v_y) \in \Delta_c \quad (10)$$

$$\forall v_y \in \text{dom}(y), \exists v_x \in \text{dom}(x) : (v_x, v_y) \in \Delta_c \quad (11)$$

dove x e y sono le due variabili del vincolo c e Δ_c è l'insieme degli assegnamenti parziali consistenti di c .

Si noti che esistono vari algoritmi di arco-consistenza che assumono che i domini delle variabili siano finiti. L'algoritmo normalmente più utilizzato è AC-3, che ha una complessità temporale asintotica di caso pessimo di classe $O(e k^3)$, dove e è il numero di vincoli binari e k è la cardinalità massima dei domini delle variabili coinvolte nei vincoli binari. Si noti, comunque, che la complessità temporale asintotica di caso pessimo di un algoritmo

di arco-consistenza ottimo è di classe $O(e k^2)$, a cui si può arrivare con l'algoritmo AC-4 mediante un significativo incremento della memoria utilizzata rispetto ad AC-3.

In più, si noti che la proprietà di arco-consistenza può essere estesa a vincoli globali introducendo la cosiddetta **iperarco-consistenza**. Infine, si noti che la nodo-consistenza coinvolge una sola variabile mentre la arco-consistenza coinvolge due variabili. Quindi è ragionevole aspettarsi di poter definire la cosiddetta **consistenza di percorso** costruendo percorsi di più di due variabili collegate da vincoli binari.

2 Problemi con Vincoli Polinomiali

La consistenza di nodo e la consistenza di arco permettono di rimuovere valori dai domini delle variabili e sono spesso efficaci, specialmente se i vincoli sono espressi in forma tabellare. Viceversa, se i vincoli sono espressi in forma intensionale, ad esempio perché i domini delle variabili non sono finiti, vengono spesso utilizzate altre forme di consistenza. Un esempio interessante a questo riguardo è rappresentato dai vincoli espressi mediante equazioni, disequazioni e disuguaglianze tra polinomi a coefficienti di variabili reali.

Si consideri un CSP \mathcal{P} con $n \in \mathbb{N}_+$ variabili in $V = \{x_i\}_{i=1}^n$ alle quali sono associati i domini $dom(x_i) = [\underline{x}_i, \overline{x}_i]$ rappresentati da intervalli chiusi in \mathbb{R} . Il problema può contenere anche più variabili, ma quelle di V sono quelle interessanti per i vincoli espressi mediante polinomi. Un vincolo c sulle variabili $V_c \subseteq V$ di \mathcal{P} si dice **polinomiale** se il suo insieme degli assegnamenti parziali consistenti può essere espresso in modo intensionale come

$$\Delta_c = \{\mathbf{x} \in dom(c) : p(\mathbf{x}) \odot q(\mathbf{x})\} \quad (12)$$

dove $\odot \in \{<, \leq, =, \neq, \geq, >\}$, $p : \mathbb{R}^k \rightarrow \mathbb{R}$ e $q : \mathbb{R}^k \rightarrow \mathbb{R}$ sono due funzioni polinomiali e $k = |V_c|$.

Si noti subito che ogni vincolo polinomiale può essere ridotto ad una delle due seguenti forme

$$\{\mathbf{x} \in dom(c) : p(\mathbf{x}) \geq 0\} \quad \text{oppure} \quad \{\mathbf{x} \in dom(c) : p(\mathbf{x}) > 0\} \quad (13)$$

utilizzando semplici manipolazioni algebriche e sfruttando il fatto che, data una funzione polinomiale $p : \mathbb{R}^k \rightarrow \mathbb{R}$, vale, per ogni $\mathbf{x} \in \mathbb{R}^k$,

$$p(\mathbf{x}) = 0 \iff p(\mathbf{x}) \leq 0 \wedge p(\mathbf{x}) \geq 0 \quad (14)$$

$$p(\mathbf{x}) \neq 0 \iff p^2(\mathbf{x}) > 0 \quad (15)$$

Infine, si noti che è sempre possibile riscrivere i vincoli in modo che tutte le variabili abbiano domini definiti come intervalli chiusi di \mathbb{R}_+ mediante opportune trasformazioni affini operate sulle variabili. Infatti, essendo tutti i domini delle variabili degli intervalli chiusi, è sufficiente traslare tutti gli intervalli per garantire che le variabili abbiano domini definiti come intervalli chiusi di \mathbb{R}_+ .

Un CSP si dice **consistente sugli intervalli** (o **bounds consistent**) se per ogni vincolo polinomiale c vale che per ognuna delle sue variabili x con dominio $[\underline{x}, \overline{x}]$ esiste almeno un assegnamento parziale consistente del vincolo che abbia il valore \underline{x} per x ed, eventualmente un altro, assegnamento parziale consistente che abbia il valore \overline{x} per x .

Quindi, indipendentemente dalla consistenza dei valori all'interno dell'intervallo $[\underline{x}, \overline{x}]$, la bounds consistency considera solo la consistenza dei valori ai due estremi di ogni intervallo coinvolto nel vincolo. Si noti che normalmente si è interessati al più piccolo intervallo, nel senso del contenimento, che garantisca la bounds consistency di una variabile in uno o più vincoli del problema.

Un algoritmo di bounds consistency è un algoritmo che trasforma un CSP in un secondo CSP ad esso equivalente in cui gli intervalli che definiscono i domini delle variabili garantiscono la bounds consistency. Come già discusso, un algoritmo di bounds consistency può considerare solo variabili definite su \mathbb{R}_+ e vincoli in forma di disequazione, stretta o meno. In queste ipotesi, è possibile definire un opportuno algoritmo di bounds consistency sfruttando la seguente proposizione.

Proposizione 1. *Dato $n \in \mathbb{N}$, sia $p : \mathbb{R}^n \rightarrow \mathbb{R}$ una funzione polinomiale a coefficienti reali positivi, se $\underline{\mathbf{x}} \in \mathbb{R}_+^n$ e $\bar{\mathbf{x}} \in \mathbb{R}_+^n$, allora per ogni $\mathbf{v} \in [\underline{\mathbf{x}}, \bar{\mathbf{x}}]$ vale*

$$p(\underline{\mathbf{x}}) \leq p(\mathbf{v}) \leq p(\bar{\mathbf{x}}) \quad (16)$$

nell'ipotesi non restrittiva che per ogni $1 \leq i \leq n$ valga $\underline{x}_i \leq \bar{x}_i$.

La descrizione generale di un algoritmo di bounds consistency che sfrutti la proposizione precedente richiede l'introduzione della notazione dei **multi-indici**. Quindi, anziché descrivere un algoritmo di bounds consistency nella sua generalità, verrà descritto sommariamente un metodo che permette di ridurre i domini delle variabili sfruttando il fatto che queste siano definite su intervalli chiusi di \mathbb{R}_+ .

Si consideri vincolo descritto dalla proprietà $p(\mathbf{x}) \geq 0$ nel caso in cui p sia una funzione lineare a coefficienti reali di $n \in \mathbb{N}_+$ variabili

$$p(\mathbf{x}) = \sum_{i=1}^n a_i x_i + b \quad (17)$$

con, per semplicità $b \geq 0$. La proprietà che definisce il vincolo può essere riscritta spostando a destra tutti i termini di p con coefficienti negativi e quindi

$$\sum_{a_i > 0} a_i x_i + b \geq \sum_{a_i < 0} -a_i x_i \quad (18)$$

Fissata una variabile di indice k , se $a_k > 0$ è possibile scrivere

$$\sum_{i \neq k, a_i > 0} a_i x_i + a_k x_k + b \geq \sum_{a_i < 0} -a_i x_i \quad (19)$$

Sfruttando la proposizione precedente è possibile ottenere

$$\sum_{i \neq k, a_i > 0} a_i x_i \leq \sum_{i \neq k, a_i > 0} a_i \bar{x}_i = u \quad l = \sum_{a_i < 0} -a_i \underline{x}_i \leq \sum_{a_i < 0} -a_i x_i \quad (20)$$

e quindi la proprietà che descrive il vincolo impone che

$$x_k \geq \frac{l - u - b}{a_k} \quad (21)$$

per ogni variabile x_k che viene moltiplicata in p per un coefficiente positivo. Si noti subito che è semplice rimuovere l'ipotesi $b \geq 0$ e che, ragionamenti simili a quelli appena visti, consentono di identificare una disequazione da utilizzare per le variabili che vengono moltiplicate per un coefficiente negativo in p . Entrambe queste disequazioni possono anche essere utilizzate per trattare vincoli descritti mediante disequazioni strette, previo cambiamento del simbolo di disuguaglianza. Si capisce quindi facilmente come la possibilità di

stimare i valori massimi e minimi di un polinomio di una variabile in un intervallo chiuso sia di fondamentale importanza per gli algoritmi di bounds consistency.

Tutte queste disequazioni possono essere utilizzate ripetutamente per restringere i domini delle variabili del vincolo considerato fino al raggiungimento di almeno una delle seguenti condizioni:

1. L'applicazione delle disequazioni non genera nuovi restringimenti dei domini e quindi è possibile cercare le soluzioni del CSP nei nuovi domini; oppure
2. L'applicazione delle disequazioni ha svuotato almeno uno dei domini e quindi è stato dimostrato che il CSP non ammette soluzioni.

I due esempi che seguono mostrano come applicare questo metodo nei due casi appena descritti. In particolare, il seguente esempio consente di restringere i domini delle variabili.

Esempio 3. Si consideri un CSP con tre variabili x , y e z i cui domini sono $dom(x) = dom(y) = dom(z) = [1, 10]$. Si consideri il vincolo definito dalla proprietà

$$2x + 3y - z \leq 1 \quad (22)$$

La proprietà che definisce il vincolo può essere riscritta come

$$z + 1 \geq 2x + 3y \quad (23)$$

e quindi

$$z + 1 \geq 2x + 3y \geq 5 \quad (24)$$

da cui si evince che $z \geq 4$. Ma, ragionando sulla variabile x si ottiene

$$11 \geq z + 1 \geq 2x + 3y \geq 2x + 3 \quad (25)$$

da cui si evince che $x \leq 4$. Infine, ragionando sulla variabile y si ottiene

$$11 \geq z + 1 \geq 2x + 3y \geq 2 + 3y \quad (26)$$

da cui si evince che $y \leq 3$. Si noti che questi ragionamenti possono essere iterati finché nessun dominio viene più modificato oppure finché almeno uno dei domini non diventa l'insieme vuoto. Nel primo caso si è ottenuto un CSP bounds consistent equivalente al CSP iniziale, mentre nel secondo caso è stato dimostrato che il CSP iniziale non ammette soluzioni. Provando ad iterare i ragionamenti fatti si nota subito che i domini non vengono ulteriormente ridotti e quindi è stato ottenuto un nuovo CSP, equivalente a quello iniziale ma bounds consistent, con tre variabili x , y e z aventi domini

$$dom(x) = [1, 4] \quad dom(y) = [1, 3] \quad dom(z) = [4, 10] \quad (27)$$

e un solo vincolo che richiede che valga la proprietà $2x + 3y - z \leq 1$. Si noti che il vincolo non può essere rimosso perché la proprietà di bounds consistency non fornisce informazioni sulla consistenza dei valori interni agli intervalli considerati. La bounds consistency studia unicamente gli estremi dei domini delle variabili.

Il seguente esempio mostra come sia possibile stabilire che il CSP considerato non ammette soluzioni. Infatti, l'applicazione delle disequazioni studiate in precedenza consente di ridurre all'insieme vuoto il dominio di almeno una variabile, terminando il processo di riduzione e stabilendo che il CSP non ammette soluzioni.

Esempio 4. Si consideri un CSP con tre variabili x , y e z i cui domini sono $dom(x) = dom(y) = dom(z) = [1, 10]$. Si consideri il vincolo definito dalla proprietà

$$12x + 8y - z \leq 1 \quad (28)$$

La proprietà che definisce il vincolo può essere riscritta come

$$z + 1 \geq 12x + 8y \quad (29)$$

e quindi

$$z + 1 \geq 12x + 8y \geq 20 \quad (30)$$

da cui si evince che il vincolo richiede che $z \geq 19$. Però, $z \leq 10$ per ipotesi e quindi è stato facilmente dimostrato che il CSP non ammette soluzioni.

Si noti che il metodo per la riduzione dei domini delle variabili descritto sommariamente nel caso di funzioni lineari può essere esteso facilmente al caso di funzioni polinomiali non lineari ricordando che, nelle ipotesi considerate, se una variabile x ha per dominio $dom(x) = [\underline{x}, \bar{x}] \subseteq \mathbb{R}_+$, allora se $m \in \mathbb{N}_+$

$$\underline{x}^{m-1} x \leq x^m \leq \bar{x}^{m-1} x \quad (31)$$

e quindi i termini non lineari del tipo x^m possono essere ridotti a termini lineari. In modo simile, se una seconda variabile y ha dominio $dom(y) = [\underline{y}, \bar{y}] \subseteq \mathbb{R}_+$, allora

$$\underline{x}y \leq xy \leq \bar{x}y \quad \text{e} \quad \underline{y}x \leq xy \leq \bar{y}x \quad (32)$$

e quindi i termini non lineari del tipo xy possono essere ridotti a termini lineari.

I vincoli polinomiali vengono spesso utilizzati restringendo i domini delle variabili a intervalli in \mathbb{Z}_+ e richiedendo che i coefficienti delle funzioni polinomi siano in \mathbb{Z} . In questo caso si parla di **vincoli polinomiali a domini finiti** ed è possibile sfruttare le particolarità di questo tipo di vincoli per definire degli algoritmi di propagazione dei vincoli efficaci ed efficienti. In particolare, il fatto che le variabili abbiano domini in \mathbb{Z}_+ e il fatto che i coefficienti delle funzioni polinomiali siano in \mathbb{Z} permette di ridurre lo studio di questo tipo di vincoli a sole disequazioni non strette perché vale

$$p(\mathbf{x}) > 0 \iff p(\mathbf{x}) - 1 \geq 0 \quad (33)$$

per una qualsiasi funzione polinomiale a coefficienti interi e variabili intere positive. In più, si noti che, siccome le variabili sono ristrette ad assumere solo valori interi, in questo caso viene adottata la notazione degli intervalli interi e si scrive $[\underline{x}..\bar{x}]$, con $\underline{x} \in \mathbb{Z}$, $\bar{x} \in \mathbb{Z}$ e $\underline{x} \leq \bar{x}$, per indicare l'intervallo contenente tutti gli interi maggiori o uguali a \underline{x} e minori o uguali a \bar{x} . Naturalmente, $[\underline{x}..\bar{x}] = \emptyset$ se $\underline{x} > \bar{x}$.

Appunti del Corso di Intelligenza Artificiale

Programmazione Logica

Prof. Federico Bergenti

10 maggio 2024

1 Paradigmi di Programmazione

Seguendo l'approccio introdotto da *Robert W. Floyd*, ogni linguaggio di programmazione può essere descritto nei termini di uno specifico **paradigma di programmazione** che ne riassume le peculiarità elencando in modo astratto le caratteristiche degli **esecutori** in grado di produrre una **computazione** partendo da un programma scritto nel linguaggio. Spesso i linguaggi di programmazione seguono più paradigmi, ma normalmente è possibile identificare un paradigma principale. Solo raramente i linguaggi di programmazione sono veramente **multi-paradigma**.

I paradigmi di programmazione che vengono tradizionalmente identificati sono:

1. Paradigma **imperativo**: in cui ogni esecutore è una macchina di Turing e un programma descrive in modo esplicito e dettagliato quali azioni deve compiere l'esecutore per risolvere un problema.
2. Paradigma **dichiarativo**: in cui ogni esecutore è in grado di trovare una soluzione ad una classe di problemi mediante una tecnica risolutiva di uso generale e un programma descrive in modo esplicito e dettagliato un problema da risolvere.

Quindi, nella programmazione imperativa, un programma descrive una procedura per risolvere una classe di problemi e l'esecutore si limita ad applicare la procedura per risolvere una specifica istanza del problema. Viceversa, nella programmazione dichiarativa, un programma descrive un problema da risolvere in modo che l'esecutore possa risolverlo mediante l'applicazione del metodo risolutivo che ha a disposizione. Usando uno slogan: *Declarative programming tells what to do. Imperative programming tells how to do it.*

Nell'ambito della programmazione imperativa, vengono normalmente identificati due paradigmi principali:

1. Paradigma **procedurale**: in cui i comandi (o, in modo improprio, **statement**) da svolgere vengono forniti ad un esecutore raggruppandoli in **procedure**. Ogni procedura manipola lo stato dell'intera computazione in modo esplicito.
2. Paradigma **object-oriented**: in cui sono presenti più esecutori detti **oggetti** che interagiscono mediante lo scambio di **messaggi**. I comandi da svolgere da parte di ogni esecutore vengono raggruppati in procedure che vengono eseguite a seguito della ricezione dei messaggi. Normalmente, ogni procedura manipola solo lo stato dell'esecutore che la esegue e non lo stato dell'intera computazione.

Nell'ambito della programmazione dichiarativa, vengono normalmente identificati due paradigmi principali:

1. Paradigma **funzionale**: in cui il problema da risolvere viene descritto mediante un insieme di oggetti e un insieme di **funzioni** tra gli oggetti (relazioni per cui ogni elemento del dominio viene associato ad un unico elemento del codominio) e un esecutore è in grado di ragionare sulle funzioni e sulla loro composizione per risolvere il problema.
2. Paradigma **logico**: in cui il problema da risolvere viene descritto mediante un insieme di oggetti e un insieme di relazioni tra gli oggetti e un esecutore è in grado di ragionare sulle relazioni per risolvere il problema.

La programmazione funzionale viene usata molto spesso nell'intelligenza artificiale, specialmente nella tradizione statunitense. I linguaggi di programmazione funzionale più tradizionali sono *Lisp* (da *LISt Processor*) e i suoi dialetti e derivati (ad esempio *Scheme*). Il linguaggio di programmazione funzionale oggi più utilizzato è sicuramente *Haskell* (www.haskell.org).

La programmazione logica viene usata molto spesso nell'intelligenza artificiale, specialmente nella tradizione europea e giapponese. I linguaggi di programmazione logica più tradizionali sono *Prolog* (da *PROgrammation en LOGique*) e i suoi dialetti e derivati. Il linguaggio di programmazione logica oggi più utilizzato è ancora Prolog, eventualmente arricchito dalle funzionalità della *programmazione logica con vincoli* (*CLP*, da *Constraint Logic Programming*), e l'implementazione più diffusa di Prolog è SWI-Prolog (www.swi-prolog.org).

Dal punto di vista dei costrutti linguistici messi a disposizione dal linguaggio, la differenza principale tra un linguaggio che segue il paradigma imperativo e un linguaggio che segue il paradigma dichiarativo è l'assenza in quest'ultimo dell'**assegnazione (o assegnamento) distruttiva**. L'assegnazione distruttiva viene utilizzata nel paradigma imperativo per consentire al programma di modificare esplicitamente lo stato della computazione. L'assenza dell'assegnazione distruttiva limita la possibilità di realizzare le comuni strutture di controllo della programmazione imperativa e, di fatto, rende la ricorsione uno strumento imprescindibile della programmazione funzionale e della programmazione logica.

Si noti che molti linguaggi seguono più di un paradigma e quindi vengono detti **multi-paradigma**. Ad esempio, il linguaggio *Kotlin* (www.kotlinlang.org) fonde in modo organico la programmazione object-oriented di Java con la programmazione funzionale ispirata ad Haskell.

Infine, si noti che il paradigma di programmazione seguito da un linguaggio non ne limita in alcun modo la potenza. Infatti, tutti i linguaggi menzionati sono **Turing equivalenti (o completi)** in quanto sono in grado di istruire i relativi esecutori a calcolare una qualsiasi funzione (Turing) computabile.

2 Il Linguaggio dei Termini

I linguaggi di programmazione dichiarativa sono spesso caratterizzati dall'uso di tipi dato strutturati che permettono la cosiddetta **programmazione simbolica** e che, quindi, rappresentano una base solida per l'intelligenza artificiale **simbolica**, che raccoglie gli approcci all'intelligenza artificiale che prevedono la manipolazione di simboli come strumento a supporto del ragionamento umano o razionale. Il linguaggio dei termini introdotto nel seguito è l'esempio più classico di tipo di dato introdotto esplicitamente per supportare la programmazione simbolica.

Si considerino due insiemi di simboli A e V non entrambi vuoti e tra loro disgiunti e, quindi, $(A \neq \emptyset \vee B \neq \emptyset) \wedge A \cap B = \emptyset$. L'insieme A viene detto insieme degli **atomi** (o dei **simboli atomici**) e l'insieme V viene detto insieme delle **variabili** (o dei **simboli di variabile**). L'insieme T dei **termini (del primo ordine)** ottenibili da A e da V è costruito secondo le regole:

1. Una variabile $v \in V$ è un termine;
2. Un atomo $a \in A$ è un termine;
3. Se $f \in A$, $k \in \mathbb{N}_+$ e $\{t_i\}_{i=1}^k \subseteq T$, allora $f(t_1, t_2, \dots, t_k)$ è un termine; e
4. Nient'altro è un termine.

Quindi, il **linguaggio dei termini (del primo ordine)** T è il linguaggio che si appoggia all'alfabeto improprio (perché non necessariamente finito) $A \cup V$ e si basa sulla seguente grammatica non contestuale:

$$\begin{aligned} Term &\rightarrow Variable \mid Atom \mid Atom(Arguments) \\ Arguments &\rightarrow Term \mid Term, Arguments \\ Variable &\rightarrow \{v \in V\} \\ Atom &\rightarrow \{a \in A\} \end{aligned} \tag{1}$$

Un termine t si dice **non strutturato** se è formato unicamente da un atomo o da una variabile. Viceversa, tutti gli altri termini si dicono **strutturati**. L'atomo più a sinistra di un termine strutturato viene detto **testa** del termine e, si noti, non può essere una variabile. Il numero di argomenti di un termine strutturato si dice **arità** del termine. Si noti che non è prevista un'arità fissa associata ad un atomo usato come testa di un termine strutturato. Quindi, ad esempio, in uno stesso termine un atomo può essere usato con arità diverse senza generare ambiguità. Per convenzione, l'arità di un termine non strutturato vale zero, ma spesso si evita di attribuire un'arità alle variabili.

In alcuni contesti, agli atomi non è attribuita un'arità e, quando viene fatta questa scelta, gli atomi vengono separati in due insiemi disgiunti chiamati, rispettivamente, insieme delle **costanti** (o dei **simboli di costante**) e insieme delle **funzioni** (o dei **simboli di funzione**) e questi insiemi vengono scelti disgiunti dall'insieme delle variabili.

Seguono alcuni esempi di termini (con $A = \{a, b, f, g\}$ e $V = \{x, y\}$):

- a e x sono termini.
- $f(a)$, $f(x)$, $f(a, f(x))$, $f(g(a))$, $f(g(y, b))$ sono termini.

Si noti che i termini strutturati possono essere usati per descrivere degli alberi e, spesso, vengono descritti mediante dei diagrammi. Ad esempio, Figura 1 riporta il diagramma che descrive l'albero relativo al termine $f(g(a, b), f(x, g(c, y)))$ (con $A = \{a, b, c, f, g\}$ e $V = \{x, y\}$).

Dato un termine $t \in T$, $vars(t)$ è l'insieme delle variabili presenti in t . Se $vars(t) = \emptyset$, allora t viene detto termine **ground**. Dato un insieme di atomi $A \neq \emptyset$, l'insieme dei termini ottenibili usando l'insieme di variabili $V = \emptyset$ è un insieme di termini ground e viene detto **universo di Herbrand** ottenibile da A .

Dati due termini $t_1 \in T$ e $t_2 \in T$, si dice che t_1 è **sintatticamente equivalente** a t_2 se t_1 e t_2 sono lo stesso elemento di T .

Quando si lavora con un insieme di termini è spesso utile permettere di trattare alcuni termini con varianti sintattiche speciali che ne semplificano la lettura e la scrittura oltre a

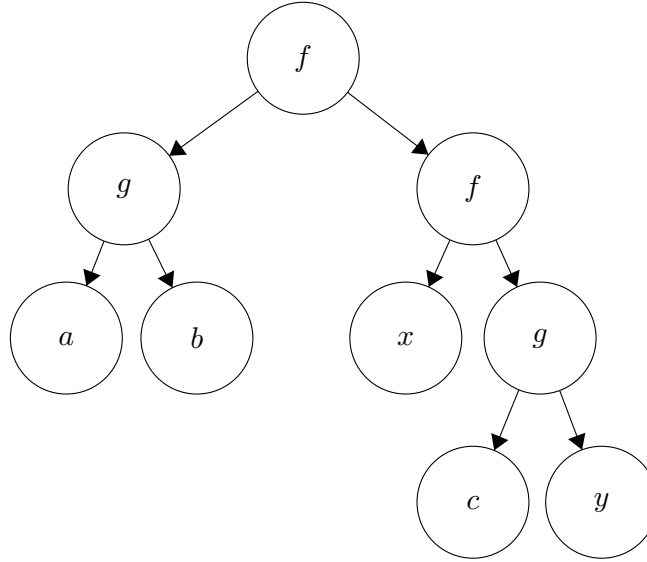


Figura 1: L'albero descritto dal termine $f(g(a, b), f(x, g(c, y)))$.

renderne più esplicito il significato. Come discusso nel seguito, queste varianti sintattiche permetteranno di scrivere termini anche complessi in modo semplice esplicitandone anche il significato. Un caso tipico di variante sintattica è quello che prevedere che l'insieme degli atomi contenga almeno due atomi:

1. Un atomo, normalmente indicato con il simbolo $.$, che viene utilizzato solo per formare termini strutturati con due argomenti; e
2. Un atomo, normalmente indicato con il simbolo $[]$, che viene utilizzato solo per formare termini non strutturati.

In questo caso, si utilizza la sintassi delle **liste** per semplificare la lettura e la scrittura di termini complessi. In particolare:

1. La lista $[]$ è un termine che viene chiamato **lista vuota**;
2. La lista $[h|r]$ è il termine $.(h, r)$, dove h è un termine che viene detto **testa della lista** e r è una lista detta **resto della lista**;
3. La lista $[t_1, t_2, \dots, t_n]$ è il termine $.(t_1, .(t_2, \dots, .(t_n, []) \dots))$; e
4. La lista $[t_1, t_2, \dots, t_n|r]$ è il termine $.(t_1, .(t_2, \dots, .(t_n, r) \dots))$, dove r è una lista che, anche in questo caso, viene detta resto della lista.

Si noti però che se, ad esempio $A = \{a, b\}$, allora il termine $[a|b]$ è un termine sintatticamente corretto ed equivale a $.(a, b)$, ma questo termine non viene considerata una lista perché si prevede che il resto di una lista sia a sua volta una lista, eventualmente vuota. L'utilizzo di termini tipo $[a|b]$ è però interessante perché permette di introdurre le **ennuple**. Infatti, un termine tipo $[a|b]$ non è altro che una coppia.

L'introduzione della sintassi delle liste non aggiunge niente all'insieme dei termini su cui si sta lavorando, ma permette di scrivere in modo semplice ed esplicito alcuni termini, quali, ad esempio (con $A = \{a, b, c\}$ e $V = \{x\}$):

- $[a, b, c]$ è la lista formata dagli atomi a , b e c , in questo ordine.
- $[a|x]$ è una lista che inizia con l'atomo a e prosegue con un resto che viene associato alla variabile x .

Oltre alle liste, viene normalmente data la possibilità di introdurre degli operatori unari o binari per descrivere in modo semplice alcuni termini complessi. Quando viene data la possibilità di utilizzare operatori per costruire termini, viene anche data la possibilità di raggruppare gli operatori e i relativi argomenti mediante le parentesi tonde, come normalmente si fa nella scrittura delle espressioni matematiche. Seguendo una tradizione consolidata, ogni operatore viene descritto mediante le seguenti tre proprietà:

1. L'atomo da utilizzare per indicare l'operatore e quindi, ad esempio, $+$ o $-$;
2. L'indice di precedenza dell'operatore; e
3. Il tipo di operatore, descritto mediante una delle sette tipologie ammesse.

L'indice di precedenza di un operatore è un numero naturale positivo con la convenzione che al crescere dell'indice di precedenza decresce la precedenza degli operatori. Infatti, viene utilizzata la convenzione che l'indice di precedenza delle variabili e degli atomi, eventualmente utilizzati come teste di termini strutturati, vale zero. In più, vale zero anche l'indice di precedenza dei termini racchiusi tra parentesi tonde.

Le sette tipologie di operatori ammesse sono descritte dalle seguenti etichette:

1. fx , fy per gli operatori **unari prefissi**;
2. xx , xy , yfx per gli operatori **binari infissi**; e
3. xf , yf per gli operatori **unari postfissi**.

Le etichette utilizzate per identificare le sette tipologie possono essere lette informalmente nel seguente modo:

1. Il simbolo f viene usato per individuare la posizione dell'operatore;
2. Il simbolo x viene letto: una variabile, un atomo, eventualmente usato come testa di un termine strutturato, un termine tra parentesi tonde o un operatore con indice di precedenza strettamente inferiore all'indice di precedenza dell'operatore che si sta descrivendo; e
3. Il simbolo y viene letto: una variabile, un atomo, eventualmente usato come testa di un termine strutturato, un termine tra parentesi tonde o un operatore con indice di precedenza minore o uguale all'indice di precedenza dell'operatore che si sta descrivendo.

Seguono alcune definizioni di operatori molto comuni indicando nelle triplette prima l'indice di precedenza, poi la tipologia e, per ultimo, l'atomo:

- $(700, xfx, <)$, $(700, xfx, =)$, $(700, xfx, >)$, $(700, xfx, >=)$, $(700, xfx, >=)$.
- $(500, yfx, +)$, $(500, yfx, -)$.
- $(400, yfx, *)$, $(400, yfx, /)$.
- $(200, fy, +)$, $(200, fy, -)$.

L'introduzione della sintassi degli operatori unari e binari non aggiunge niente all'insieme dei termini su cui si sta lavorando ma permette di scrivere in modo semplice ed esplicito alcuni termini complessi. Seguono alcuni esempi dell'uso dei precedenti operatori per la costruzione di termini (con $A = \{a, b, c, f\}$ e $V = \{x, y\}$) in cui è stata usata la convenzione comune di racchiudere un atomo tra apici per garantire che non venga interpretato come un operatore:

1. Il termine $f(x) + b$ equivale a $' + '(f(x), b)$;
2. Il termine $a + b - c$ equivale a $' - '(' + '(a, b), c)$;
3. Il termine $a + b * c$ equivale a $' + '(a, ' * '(b, c))$;
4. Il termine $-a + b/y$ equivale a $' + '(' - '(a), '/'(b, y))$
5. Il termine $a + b =< c * a$ equivale a $' =< '(' + '(a, b), ' * '(c, a))$.

Si noti che l'introduzione contemporanea di liste e operatori viene normalmente accompagnata dalla definizione dell'operatore $.$ come operatore unario postfisso con basso indice di precedenza (normalmente, 100). In questo caso, l'atomo $.$ utilizzato per le liste deve essere racchiuso tra apici per evitare che venga interpretato come operatore unario postfisso.

Si noti che le informazioni fornite per descrivere gli operatori consentono di identificare in modo univoco a quale termine si sta facendo riferimento utilizzando gli operatori, purché non vengano introdotti degli errori sintattici. L'introduzione degli operatori aumenta la possibilità di introdurre errori sintattici perché le sette tipologie di operatori vincolano i modi in cui gli operatori possono essere combinati. Ad esempio, i seguenti sono alcuni esempi di errori sintattici dovuti all'introduzione degli operatori per descrivere termini:

1. Il termine $a + b >=$ utilizza l'operatore $>=$ come unario postfisso anziché binario;
2. Il termine $a =< b =< c$ utilizza l'operatore $=<$ con un secondo operatore con lo stesso indice di precedenza a sinistra (o a destra); e
3. Il termine $(a + b)$ non ha le parentesi bilanciate.

Infine, conviene notare che anche se l'introduzione di operatori per usi specifici può semplificare la manipolazione di termini complessi, non conviene ricorrere troppo spesso a questi operatori.

3 Unificazione di Termini

Dato un insieme di termini T costruito su un insieme di atomi A e un insieme di variabili V , una **sostituzione** è un insieme finito ed eventualmente vuoto della forma

$$\{t_1/x_1, t_2/x_2, \dots, t_n/x_n\} \quad (2)$$

dove

1. x_1, x_2, \dots, x_n sono variabili;
2. t_1, t_2, \dots, t_n sono termini;
3. Per ogni $1 \leq i \leq n$, $t_i \neq x_i$; e
4. Per ogni $1 \leq i \leq n$ e $1 \leq j \leq n$, se $i \neq j$ allora $x_i \neq x_j$.

Se t è un termine e θ è una sostituzione, allora $t\theta$ è il termine che si ottiene sostituendo in t *simultaneamente* tutte le variabili nelle parti destre degli elementi della sostituzione θ con i rispettivi termini.

Ad esempio, se $\theta = \{f(z, z)/x, c/z\}$ (con $A = \{c, f\}$ e $V = \{x, z\}$) allora

$$p(f(x, y), x, g(z))\theta = p(f(f(z, z), y), f(z, z), g(c)) \quad (3)$$

Date due sostituzioni $\theta = \{t_1/x_1, t_2/x_2, \dots, t_n/x_n\}$ e $\sigma = \{u_1/y_1, u_2/y_2, \dots, u_m/y_m\}$ la **sostituzione composta** $\theta \circ \sigma$ si ottiene dal seguente insieme

$$\{t_1\sigma/x_1, t_2\sigma/x_2, \dots, t_n\sigma/x_n, u_1/y_1, u_2/y_2, \dots, u_m/y_m\} \quad (4)$$

eliminando tutti gli elementi

1. $t_j\sigma/x_j$ se vale $t_j\sigma = x_j$; e
2. u_j/y_j se $y_j \in \{x_1, x_2, \dots, x_n\}$.

Quindi, ad esempio, se $\theta = \{f(y)/x, z/y\}$ e $\sigma = \{a/x, b/y, y/z\}$ (con $A = \{a, b, f\}$ e $V = \{x, y, z\}$) allora

$$\theta \circ \sigma = \{f(b)/x, y/z\} \subset \{f(b)/x, y/y, a/x, b/y, y/z\} \quad (5)$$

La proprietà fondamentale della composizione di sostituzioni, che ne giustifica il nome, è che per ogni coppia di sostituzioni θ, σ vale

$$t(\theta \circ \sigma) = (t\theta)\sigma \quad (6)$$

per ogni termine t . Quindi, ad esempio, se $\theta = \{f(y)/x, z/y\}$, $\sigma = \{a/x, b/y, y/z\}$ e $t = h(x, g(y), z)$ (con $A = \{a, b, f, g, h\}$ e $V = \{x, y, z\}$), allora

$$t\theta = h(f(y), g(z), z) \quad (t\theta)\sigma = h(f(b), g(y), y) \quad (7)$$

e, infatti, $t(\theta \circ \sigma) = h(f(b), g(y), y)$.

Una sostituzione θ si dice **unificatore** per l'insieme di $k \geq 2$ termini $S = \{t_i\}_{i=1}^k$ se vale la seguente catena di uguaglianze

$$t_1\theta = t_2\theta = \dots = t_k\theta \quad (8)$$

Un insieme di termini si dice **unificabile** se esiste almeno un unificatore per l'insieme. Si noti che un insieme di termini può ammettere più unificatori, ad esempio, perché alcuni unificatori utilizzano più variabili del necessario. L'insieme di due termini $\{p(x), p(y)\}$ (con $A = \{p\}$ e $V = \{x, y, z\}$) ammette almeno tre unificatori:

$$\{x/y\} \quad \{y/x\} \quad \{z/x, z/y\} \quad (9)$$

dove i primi due unificatori possono essere resi uguali semplicemente cambiando in modo coerente i nomi delle variabili e il terzo unificatore è stato ottenuto mediante l'introduzione della variabile, non necessaria, z .

Viceversa, se un insieme di termini S è unificabile allora ammette un unico **unificatore più generale** (*Most General Unifier*, **MGU**), a meno di cambiamenti coerenti dei nomi delle variabili. Il MGU di un insieme di termini S è definito come un unificatore $mgu(S)$ tale che qualsiasi sia un unificatore θ di S esiste almeno una sostituzione σ tale che $\theta = mgu(S) \circ \sigma$. Quindi, a giustificazione del nome, il MGU di un insieme di termini non compie

sostituzioni non necessarie ma compie solo le sostituzioni necessarie a rendere gli elementi dell'insieme congiuntamente uguali. Dei tre unificatori elencati nell'esempio precedente, solo i primi due sono MGU e, infatti, sono uguali a meno di cambiamenti coerenti dei nomi delle variabili.

Ad esempio, se $S = \{p(x), p(f(y))\}$ allora $\theta = \{f(a)/x, a/y\}$ (con $A = \{a, f, p\}$ e $V = \{x, y\}$) è un unificatore di S , infatti

$$p(x)\theta = p(f(a)) \quad p(f(y))\theta = p(f(a)) \quad (10)$$

ma θ non è il MGU di S perché, per unificare gli elementi di S , è sufficiente $\alpha = \{f(y)/x\}$ e, infatti, $\theta = \alpha \circ \{a/y\}$.

Dato un insieme di termini T costruito su un insieme di atomi A e un insieme di variabili V , un **problema di unificazione (sintattica)** è un insieme del tipo

$$\{l_1 \doteq r_1, l_2 \doteq r_2, \dots, l_n \doteq r_n\} \quad (11)$$

dove $\{l_i\}_{i=1}^n \subseteq T$ e $\{r_i\}_{i=1}^n \subseteq T$ e ogni elemento del problema viene detto **equazione**. Dato un problema di unificazione, si cerca una sostituzione θ tale che valga $l_i\theta = r_i\theta$ per ogni $1 \leq i \leq n$. Se una tale sostituzione esiste, allora il problema viene detto risolubile e la sostituzione trovata ne è una soluzione.

Si noti che vengono definite le seguenti funzioni per un problema di unificazione S , generalizzando opportunamente le relative funzioni definite per i termini:

1. $vars(S)$ è l'insieme delle variabili contenute nelle parti sinistre e nelle parti destre delle equazioni di S ; e
2. Se θ è una sostituzione, $S\theta$ è il problema di unificazione ottenuto applicando la sostituzione alle parti sinistre e alle parti destre di tutte le equazioni in S .

Il seguente algoritmo è in grado di stabilire se un problema di unificazione è risolubile e, in caso, di trovarne una soluzione. L'algoritmo (**di unificazione di Martelli e Montanari**) parte da un problema di unificazione S e termina producendo uno dei seguenti risultati:

1. Il simbolo \perp se il problema di unificazione non è risolubile; oppure
2. Un problema di unificazione equivalente ad S ma con solo variabili nelle parti sinistre delle equazioni, e quindi nella forma $\{x_1 \doteq t_1, x_2 \doteq t_2, \dots, x_m \doteq t_m\}$, da cui è possibile costruire una soluzione $\{t_1/x_1, t_2/x_2, \dots, t_m/x_m\}$.

L'algoritmo può essere descritto come la seguente **funzione non deterministica** *unify*:

1. $unify(G \cup \{f(l_1, l_2, \dots, l_m) \doteq f(r_1, r_2, \dots, r_m)\}) = unify(G \cup \{l_1 \doteq r_1, l_2 \doteq r_2, \dots, l_m \doteq r_m\})$ (caso *decompose*);
2. $unify(G \cup \{x \doteq t\}) = unify(G \cup \{t/x\} \cup \{x \doteq t\})$ se $x \in vars(G)$ e $x \notin vars(t)$ (caso *eliminate*);
3. $unify(G \cup \{f(l_1, l_2, \dots, l_m) \doteq x\}) = unify(G \cup \{x \doteq f(l_1, l_2, \dots, l_m)\})$ se $x \in V$ (caso *swap*);
4. $unify(G \cup \{t \doteq t\}) = unify(G)$ (caso *delete*);
5. $unify(G \cup \{f(l_1, l_2, \dots, l_m) \doteq g(r_1, r_2, \dots, r_k)\}) = \perp$ se $f \neq g \vee m \neq k$ (caso *conflict*);
6. $unify(G \cup \{x \doteq f(t_1, t_2, \dots, t_m)\}) = \perp$ se $x \in vars(f(t_1, t_2, \dots, t_m))$ (caso *check*).

Si noti che l'algoritmo è utilizzabile per calcolare il MGU di due termini l e r tra loro unificabili. Infatti, l'applicazione dell'algoritmo al problema di unificazione $\{l \doteq r\}$ consente di trovare immediatamente $mgu(\{l, r\})$.

In più, si noti che il caso *check* aumenta la complessità computazionale temporale asintotica di caso pessimo dell'algoritmo e quindi, alle volte, si preferisce omettere questo caso per ottenere una procedura di unificazione *senza occurs check*. In questo caso è anche necessario rimuovere le restrizioni dal caso eliminate per garantire di potere elaborare problemi tipo $\{x \doteq f(x)\}$ e, quindi, non sarà più possibile garantire la terminazione della procedura (e non più algoritmo) di unificazione.

Infine, si noti che i problemi di unificazione possono essere generalizzati a **problemi di unificazione modulo teoria** se la relazione \doteq ammette delle proprietà aggiuntive fornite dalla teoria considerata oltre ad essere una relazione di equivalenza tra termini. In questi casi, l'algoritmo visto deve essere opportunamente esteso per tenere conto delle proprietà fornite dalla teoria considerata. Ad esempio, se l'atomo f denota un'operazione commutativa della teoria considerata, allora il problema di unificazione $\{f(a, b) \doteq f(x, y)\}$ dovrà ammettere due soluzioni $\{a/x, b/y\}$ oppure $\{b/x, a/y\}$. Quindi, non solo si dovrà modificare l'algoritmo di unificazione per tenere conto della commutatività dell'operazione denotata da f ma si dovrà anche ammettere la possibilità che due termini unificabili possano avere più MGU tra loro effettivamente diversi.

4 Sintassi di Prolog₀

Si consideri un insieme di atomi A e un insieme di variabili V tali che:

1. A è formato da tutte e sole le parole che iniziano con una lettera minuscola e proseguono con, eventualmente zero, lettere, numeri e underscore; e
2. V è formato da tutte e sole le parole che iniziano con una lettera maiuscola o un underscore e proseguono con, eventualmente zero, lettere, numeri e underscore senza però essere formate unicamente da underscore.

Si noti che $A \neq \emptyset$, $V \neq \emptyset$, $A \cap V = \emptyset$ ed entrambi gli insiemi sono infiniti numerabili. In più, si noti che non sono presenti simboli di punteggiatura nei due insiemi.

Si consideri il linguaggio *Prolog₀* descritto dalla seguente grammatica che usa l'alfabeto improprio $A \cup V$ e riprende la grammatica dei termini nella parte finale:

$$\begin{aligned}
\textit{Program} &\rightarrow \textit{Clauses} \\
\textit{Clauses} &\rightarrow \textit{Clause} \mid \textit{Clause Clauses} \\
\textit{Clause} &\rightarrow \textit{Fact} \mid \textit{Rule} \\
\textit{Fact} &\rightarrow \textit{Head}. \\
\textit{Rule} &\rightarrow \textit{Head} \textit{:}- \textit{Body}. \\
\textit{Head} &\rightarrow \textit{Atom} \mid \textit{Atom}(\textit{Arguments}) \\
\textit{Body} &\rightarrow \textit{Conjuncts} \\
\textit{Conjuncts} &\rightarrow \textit{Conjunct} \mid \textit{Conjunct}, \textit{Conjuncts} \\
\textit{Conjunct} &\rightarrow \textit{Atom} \mid \textit{Atom}(\textit{Arguments}) \\
\textit{Term} &\rightarrow \textit{Variable} \mid \textit{Atom} \mid \textit{Atom}(\textit{Arguments}) \\
\textit{Arguments} &\rightarrow \textit{Term} \mid \textit{Term}, \textit{Arguments} \\
\textit{Variable} &\rightarrow \{v \in V\} \\
\textit{Atom} &\rightarrow \{a \in A\}
\end{aligned} \tag{12}$$

Il linguaggio Prolog₀ è un dialetto minimalista (ma non troppo) del linguaggio Prolog. In particolare, il linguaggio Prolog₀ isola la porzione di Prolog che realizza la programmazione logica in modo più puro.

La seguente nomenclatura è definita per Prolog₀ ma vale anche per Prolog:

1. Gli elementi derivabili dalla produzione *Program*, e quindi gli elementi π tali che $Program \xrightarrow{*} \pi$, si chiamano **programmi**.
2. Gli elementi derivabili dalla produzione *Clause*, e quindi gli elementi γ tali che $Clause \xrightarrow{*} \gamma$, si chiamano **clausole definite**.
3. Gli elementi derivabili dalla produzione *Fact*, e quindi gli elementi λ tali che $Fact \xrightarrow{*} \lambda$, si chiamano **fatti** e, per ogni fatto, è definita una **testa**.
4. Gli elementi derivabili dalla produzione *Rule*, e quindi gli elementi ρ tali che $Rule \xrightarrow{*} \rho$, si chiamano **regole** e, per ogni regola, è definita una **testa** ed è definito un **corpo**.
5. Gli elementi derivabili dalla produzione *Conjunct*, e quindi gli elementi α tali che $Conjunct \xrightarrow{*} \alpha$, si chiamano **congiunti** e ogni congiunto è un termine che non può essere una variabile isolata.

Infine, si noti che gli elementi derivabili da *Conjuncts* sono sequenze di congiunti e quindi è possibile estendere ad un'intera sequenza una qualsiasi operazione che coinvolga un congiunto semplicemente applicando l'operazione ad ogni congiunto della sequenza e separando i risultati con delle virgole. L'utilizzo delle virgole per separare i risultati consente di ottenere ancora un elemento derivabile da *Conjuncts*. In particolare, se θ è una sostituzione, e $Conjuncts \xrightarrow{*} \beta$, allora è possibile parlare di $\beta \theta$ applicando la sostituzione ad ogni congiunto di β e separando i risultati ottenuti con delle virgole.

Dato un programma scritto in Prolog₀ è possibile ottenere una computazione abbinando al programma un **goal**. Una computazione così ottenuta, se produce un risultato, produce una o più sostituzioni, eventualmente vuote, che coinvolgono un sottoinsieme delle variabili del goal. Non tutte le computazioni, però, producono un risultato perché, per alcune computazioni, il risultato non è definito mentre, per altre computazioni, non necessariamente diverse dalle precedenti, la computazione non termina.

Dal punto di vista sintattico, un goal è descritto da una regola privata della testa. Un elemento dell'insieme delle clausole definite unito all'insieme dei goal viene detto **clausola di Horn**. Si noti che i goal non possono essere utilizzati nel testo di un programma Prolog₀ mentre possono essere utilizzati nel testo di un programma Prolog con l'ovvio significato di attivare una computazione per ogni goal incontrato durante il caricamento in memoria del programma.

Ad esempio, si consideri il seguente programma *Prolog₀*:

```
m(a).
m(b).

f(c).
f(d).

c(X) :- m(X).
c(X) :- f(X).
```

Questo programma può essere utilizzato con i seguenti goal, com'è semplice verificare utilizzando SWI-Prolog o la sua interfaccia Web *SWISH* (swish.swi-prolog.org). Si

noti che, per fornire un goal all'interfaccia interattiva di SWI-Prolog o a SWISH, deve essere omesso il simbolo `:-` iniziale. In più, si noti che i risultati successivi al primo vengono ottenuti premendo `;` quando si usa l'interfaccia interattiva di SWI-Prolog mentre vengono ottenuti con gli appositi pulsanti grafici quando si usa SWISH. Infine, si noti che viene stampato **true** quando il risultato della computazione è la sostituzione \emptyset e viene stampato **false** quando non è definito un risultato per la computazione.

```
:- m(a).
true

:- m(b).
true

:- m(w).
false.

:- f(X).
X=c ; X=d

:- c(X).
X=a ; X=b ; X=c ; X=d
```

La sintassi del linguaggio Prolog₀ può essere estesa sfruttando la possibilità di introdurre liste e operatori come varianti sintattiche del linguaggio dei termini, eventualmente estendendo l'insieme degli atomi in modo opportuno. Infatti, il linguaggio Prolog₀⁺ estende Prolog₀ mediante l'introduzione di liste e operatori nelle teste di fatti e regole, nei congiunti e nei termini. Ovviamente, anche i goal da utilizzare con i programmi Prolog₀⁺ potranno sfruttare liste e operatori.

Ad esempio, il seguente è un programma scritto in Prolog₀⁺ sfruttando le liste:

```
m(H, [H | X]).
m(H, [Y | R]) :- m(H, R).
```

Il programma può essere utilizzato, ad esempio, con il goal `m(X, [a, b, c])`.

Si noti che l'introduzione di liste e operatori non estende in modo significativo il linguaggio Prolog₀ perché liste e operatori non sono altro che varianti sintattiche del linguaggio dei termini. Viceversa, le due seguenti estensioni introdotte in Prolog₀⁺ non si limitano a varianti sintattiche, ma hanno anche un aspetto semantico. La prima di queste estensioni prevede che sia sempre disponibile in Prolog₀⁺ un operatore definito dalla tripla $(700, xfx, =)$, richiedendo quindi che l'atomo `=` sia presente nell'insieme degli atomi di Prolog₀⁺. La semantica di questo operatore verrà discussa in seguito. La seconda di queste estensioni prevede che l'insieme delle variabili di Prolog₀⁺ contenga anche un elemento formato unicamente da un underscore, elemento che è esplicitamente escluso dall'insieme delle variabili di Prolog₀. Questa variabile, che viene detta **dummy** (o **muta**), ha un significato particolare in quanto si prevede che venga sostituita con una nuova variabile non presente nel programma o nel goal tutte le volte che viene incontrata.

Ad esempio, l'utilizzo della variabile dummy e dell'operatore `=` permette di riscrivere il programma dell'esempio precedente come segue:

```
m(H, [X | _]) :- X = H.
m(H, [_ | R]) :- m(H, R).
```

5 Semantica di Prolog₀

La *sintassi* di Prolog₀ è definita dalla grammatica che viene utilizzata per costruire l'insieme dei programmi partendo dall'insieme degli atomi A e dall'insieme delle variabili V . La *semantica* di Prolog₀ viene definita nel seguito esplicitando formalmente la computazione che si ottiene da un programma utilizzato insieme ad un opportuno goal.

La sintassi di Prolog₀⁺ viene definita mediante semplici regole sintattiche che permettono di rimuovere liste, operatori e variabili dummy. Quindi, la semantica di Prolog₀⁺ può essere espressa, ad eccezione che per la semantica dell'operatore $=$, sfruttando la semantica dei programmi e dei relativi goal scritti in Prolog₀ dopo la rimozione di operatori, liste e variabili dummy. Quindi, dopo aver definito in modo formale la semantica di Prolog₀ sarà sufficiente definire la semantica dell'operatore $=$ per completare la semantica di Prolog₀⁺.

Dato un programma π scritto in Prolog₀ e un goal ν , la definizione formale della semantica del programma π quando viene utilizzato per **soddisfare** il goal ν viene descritta mediante una funzione non deterministica σ^π . La funzione σ^π riceve come unico argomento il goal ν e produce, in modo non deterministico, zero o più sostituzioni che coinvolgono le variabili di ν . Se almeno una sostituzione viene calcolata, anche se vuota, allora il goal si dice **soddisfacibile**. Viceversa, il goal si dice **insoddisfacibile**. Si noti, però, che non è possibile garantire che il calcolo di $\sigma^\pi(\nu)$ termini per qualsiasi coppia programma π e goal ν e, quindi, è possibile che in alcune situazioni l'enumerazione dei risultati del calcolo di $\sigma^\pi(\nu)$ possa non essere completata.

Prima di dettagliare σ^π è però necessario introdurre alcune funzioni che verranno utilizzate nella definizione di σ^π . Dato un programma π scritto in Prolog₀, la funzione $head_P(\pi)$ produce la prima clausola definita del programma, che è sempre presente visto che la grammatica di Prolog₀ non ammette programmi vuoti. In più, la funzione $rest_P(\pi)$ produce π privato della prima clausola definita, se il programma contiene almeno due clausole definite, oppure \perp se il programma è formato unicamente da una clausola definita.

In modo simile, dato un goal ν scritto in Prolog₀, la funzione $head_G(\nu)$ produce il primo congiunto di ν , che è sempre presente visto che la grammatica delle regole di Prolog₀ non ammette regole senza corpo. In più, la funzione $rest_G(\nu)$ produce \perp se il goal è formato unicamente da un congiunto, oppure la sequenza dei congiunti di ν a cui è stato rimosso il primo congiunto, separando i congiunti con delle virgole.

Infine, la funzione non deterministica $(.)'$, data una clausola definita, produce una nuova clausola definita, detta sua **variante fresh (o fresca)**, ottenuta applicando una sostituzione che associa ad ogni variabile della clausola definita una nuova variabile non ancora utilizzata nella computazione. Quindi, ad esempio, una variante fresh del fatto $h(X, Y, X)$. è

$$(h(X, Y, X).) ' = h(X1, Y1, X1). \quad (13)$$

nell'ipotesi che le variabili $X1$ e $Y1$ non siano ancora state usate nella computazione. In modo simile, una variante fresh della regola $h(X, Y) :- f(X), g(Y, X)$. è

$$(h(X, Y) :- f(X), g(Y, X).) ' = h(X1, Y1) :- f(X1), g(Y1, X1). \quad (14)$$

sempre nell'ipotesi che le variabili $X1$ e $Y1$ non siano ancora state usate nella computazione. Si noti che la funzione $(.)'$ è una funzione non deterministica perché la scelta delle nuove variabili è del tutto arbitraria, purché queste non siano ancora state utilizzate nella computazione. In più, si noti che la produzione della variante fresh di una regola richiede di memorizzare le variabili utilizzate durante la computazione. Però, per non appesantire troppo la notazione, si preferisce lasciare implicito l'insieme delle variabili utilizzate durante una computazione senza, comunque, perdere di generalità o di correttezza.

Dato un programma π scritto in Prolog₀ e un goal ν , la funzione non deterministica σ^π può essere esplicitata come segue:

$$\sigma^\pi(\nu) = \begin{cases} \sigma_G^\pi(g, \pi) & \text{se } g = \text{head}_G(\nu) \wedge \perp = \text{rest}_G(\nu) \\ \theta \circ \sigma^\pi(\text{:- } r\theta.) & \text{se } g = \text{head}_G(\nu) \wedge r = \text{rest}_G(\nu) \wedge r \neq \perp \wedge \theta = \sigma_G^\pi(g, \pi) \end{cases}$$

Quindi, il calcolo di $\sigma^\pi(\nu)$ si riduce a due casi non deterministici disgiunti. Se il goal è formato da un unico congiunto, allora viene usata la funzione non deterministica σ_G^π descritta nel seguito. Questa funzione, se è applicabile agli specifici argomenti e se termina, produce la sostituzione cercata. La stessa funzione viene utilizzata per produrre una sostituzione nel caso in cui il goal sia formato da almeno due congiunti. In questo caso, la sostituzione ottenuta da σ_G^π viene applicata alla parte non ancora elaborata del goal per applicare poi ricorsivamente σ^π . Si noti che la natura non deterministica di σ^π deriva unicamente dalla natura non deterministica di σ_G^π perché la sintassi del linguaggio garantisce che i due casi considerati nella definizione di σ^π siano disgiunti.

La funzione σ_G^π riceve come primo argomento un congiunto e come secondo argomento un programma, cioè una sequenza di clausole definite. Se effettivamente applicabile agli argomenti e nell'ipotesi che termini, σ_G^π produce in modo non deterministico delle sostituzioni. In particolare, la funzione σ_G^π è definita come segue:

$$\sigma_G^\pi(g, \pi) = \begin{cases} \sigma_F(g, c) & \text{se } c = \text{head}_P(\pi) \wedge c = h. \\ \sigma_R^\pi(g, c) & \text{se } c = \text{head}_P(\pi) \wedge c = h \text{ :- } b. \\ \sigma_G^\pi(g, r) & \text{se } r = \text{rest}_P(\pi) \wedge r \neq \perp \end{cases} \quad (15)$$

La funzione è definita da tre casi non deterministici. Il primo descrive il comportamento della funzione se la prima clausola definita è un fatto. Il secondo descrive il comportamento della funzione se la prima clausola definita è una regola. L'ultimo caso descrive il comportamento della funzione se non sono state elaborate tutte le clausole definite del programma. Quindi, i due casi più interessanti sono il primo e il secondo perché il terzo si limita a proseguire il calcolo nel tentativo di coinvolgere tutte le clausole definite del programma.

La funzione σ_F riceve come primo argomento un congiunto di un goal e come secondo argomento un fatto è definita come segue:

$$\sigma_F(g, f) = \theta \quad \text{se } f' = h'. \wedge \theta = \text{mgu}(\{g, h'\}) \wedge \theta \neq \perp \quad (16)$$

Quindi, se è possibile unificare la testa del fatto con il goal, allora il MGU ottenuto dal goal e dalla testa di una variante fresh del fatto ricevuto come argomento è il risultato cercato.

In modo simile, la funzione σ_R^π riceve come primo argomento un congiunto di un goal e come secondo argomento una regola è definita come segue:

$$\sigma_R^\pi(g, r) = \theta \circ \sigma^\pi(\text{:- } b'\theta.) \quad \text{se } r' = h' \text{ :- } b'. \wedge \theta = \text{mgu}(\{g, h'\}) \wedge \theta \neq \perp \quad (17)$$

Quindi, se è possibile unificare la testa della regola con il goal, allora il MGU ottenuto dal goal e dalla testa di una variante fresh della regola ricevuta come argomento è una parte del risultato cercato. Infatti, una volta identificato il MGU, lo si applica al corpo della variante fresh della regola utilizzata in modo da ottenere un nuovo goal da soddisfare mediante l'applicazione di σ^π .

Come già chiarito, la semantica di Prolog₀⁺ è ottenuta mediante la semantica di Prolog₀ dopo aver rimosso liste, operatori e variabili dummy in modo puramente sintattico. Per

completare la semantica di Prolog_0^+ è però necessario definire una semantica per l'operatore $=$, che è sempre ritenuto disponibile. La semantica di questo operatore può essere definita prevedendo che venga aggiunto un **prologo** ad ogni programma Prolog_0^+ . Questo prologo contiene la definizione del comportamento dell'operatore $=$ mediante il fatto $X = X$. che, appunto, stabilisce che l'operatore $=$ faccia riferimento all'uguaglianza sintattica.

Il seguente esempio mostra come sia possibile utilizzare Prolog_0 per dimostrare che un goal non è soddisfacibile.

Esempio 1. Dato il programma $\pi = p(a). p(b).$, si vuole verificare se il goal $\nu = :- p(c)$. è soddisfacibile. In particolare,

$$\begin{aligned}\sigma^\pi(\nu) &\stackrel{1}{=} \sigma_G^\pi(p(c), \pi) \stackrel{1}{=} \sigma_F(p(c), p(a).) = \times \quad \text{mgu}(\{p(c), p(a)\}) = \perp \\ \sigma^\pi(\nu) &\stackrel{1}{=} \sigma_G^\pi(p(c), \pi) \stackrel{2}{=} \times \quad \text{non applicabile} \\ \sigma^\pi(\nu) &\stackrel{1}{=} \sigma_G^\pi(p(c), \pi) \stackrel{3}{=} \sigma_G^\pi(p(c), p(b).) \stackrel{1}{=} \sigma_F(p(c), p(b).) = \times \quad \text{mgu}(\{p(c), p(b)\}) = \perp \\ \sigma^\pi(\nu) &\stackrel{1}{=} \sigma_G^\pi(p(c), \pi) \stackrel{3}{=} \sigma_G^\pi(p(c), p(b).) \stackrel{2,3}{=} \times \quad \text{non applicabili} \\ \sigma^\pi(\nu) &\stackrel{2}{=} \times \quad \text{non applicabile}\end{aligned}$$

quindi il goal non è soddisfacibile.

Il seguente esempio mostra come sia possibile utilizzare Prolog_0 per trovare una sostituzione che soddisfa un goal.

Esempio 2. Dato il programma $\pi = p(a). p(b).$, si vuole verificare se il goal $\nu = :- p(b)$. è soddisfacibile. In particolare,

$$\begin{aligned}\sigma^\pi(\nu) &\stackrel{1}{=} \sigma_G^\pi(p(b), \pi) \stackrel{1}{=} \sigma_F(p(b), p(a).) = \times \quad \text{mgu}(\{p(b), p(a)\}) = \perp \\ \sigma^\pi(\nu) &\stackrel{1}{=} \sigma_G^\pi(p(b), \pi) \stackrel{2}{=} \times \quad \text{non applicabile} \\ \sigma^\pi(\nu) &\stackrel{1}{=} \sigma_G^\pi(p(b), \pi) \stackrel{3}{=} \sigma_G^\pi(p(b), p(b).) \stackrel{1}{=} \sigma_F(p(b), p(b).) = \text{mgu}(\{p(b), p(b)\}) = \emptyset\end{aligned}$$

quindi il goal è soddisfacibile perché è stata trovata una sostituzione.

Il seguente esempio mostra come sia possibile utilizzare Prolog_0 per trovare più sostituzioni che soddisfano un goal.

Esempio 3. Dato il programma $\pi = p(a). p(b).$, si vuole verificare se il goal $\nu = :- p(X)$. è soddisfacibile. In particolare,

$$\sigma^\pi(\nu) \stackrel{1}{=} \sigma_G^\pi(p(X), \pi) \stackrel{1}{=} \sigma_F(p(X), p(a).) = \text{mgu}(\{p(X), p(a)\}) = \{a/X\} \quad (18)$$

quindi il goal è soddisfacibile. Continuando con l'applicazione non deterministica di σ^π si ottiene:

$$\begin{aligned}\sigma^\pi(\nu) &\stackrel{1}{=} \sigma_G^\pi(p(X), \pi) \stackrel{2}{=} \times \quad \text{non applicabile} \\ \sigma^\pi(\nu) &\stackrel{1}{=} \sigma_G^\pi(p(X), \pi) \stackrel{3}{=} \sigma_G^\pi(p(X), p(b).) \stackrel{1}{=} \sigma_F(p(X), p(b).) = \{b/X\}\end{aligned}$$

e quindi il goal è soddisfacibile anche dalla sostituzione $\{b/X\}$.

Il seguente esempio mostra come sia possibile utilizzare Prolog₀ per trovare una sostituzione che soddisfa un goal.

Esempio 4. Dato il programma $\pi = p(a). \ q(X) :- p(X).$, si vuole verificare se il goal $\nu = :- q(X).$ è soddisfacibile. In particolare,

$$\begin{aligned} \sigma^\pi(\nu) &\stackrel{1}{=} \sigma_G^\pi(q(X), \pi) \stackrel{1}{=} \sigma_F^\pi(q(X), p(a).) = \times \quad mgu(\{q(X), p(a)\}) = \perp \\ \sigma^\pi(\nu) &\stackrel{1}{=} \sigma_G^\pi(q(X), \pi) \stackrel{2}{=} \times \quad \text{non applicabile} \\ \sigma^\pi(\nu) &\stackrel{1}{=} \sigma_G^\pi(q(X), \pi) \stackrel{3}{=} \sigma_G^\pi(q(X), q(X) :- p(X).) \stackrel{1}{=} \times \quad \text{non applicabile} \\ \sigma^\pi(\nu) &\stackrel{1}{=} \sigma_G^\pi(q(X), \pi) \stackrel{3}{=} \sigma_G^\pi(q(X), q(X) :- p(X).) \stackrel{2}{=} \sigma_R^\pi(q(X), q(X) :- p(X).) \end{aligned}$$

ma

$$\sigma_R^\pi(q(X), q(X) :- p(X).) = \{X/X1\} \circ \sigma^\pi(:- p(X1)\{X/X1\}.) = \{X/X1\} \circ \sigma^\pi(:- p(X).)$$

e

$$\sigma^\pi(:- p(X).) \stackrel{1}{=} \sigma_G^\pi(p(X), \pi) \stackrel{1}{=} \sigma_F^\pi(p(X), p(a).) = mgu(\{p(X), p(a)\}) = \{a/X\}$$

e quindi il goal è soddisfacibile perché è stata trovata la sostituzione $\{a/X1, a/X\}$.

Il seguente esempio mostra come Prolog₀ possa essere utilizzato per generare computazioni che non terminano.

Esempio 5. Dato il programma $\pi = q(X) :- q(X).$, si vuole verificare se il goal $\nu = :- q(X).$ è soddisfacibile. In particolare,

$$\begin{aligned} \sigma^\pi(\nu) &\stackrel{1}{=} \sigma_G^\pi(q(X), \pi) \stackrel{1}{=} \times \quad \text{non applicabile} \\ \sigma^\pi(\nu) &\stackrel{1}{=} \sigma_G^\pi(q(X), \pi) \stackrel{2}{=} \sigma_R^\pi(q(X), q(X) :- q(X).) \end{aligned}$$

ma

$$\sigma_R^\pi(q(X), q(X) :- q(X).) = \{X/X1\} \circ \sigma^\pi(:- q(X1)\{X/X1\}.) = \{X/X1\} \circ \sigma^\pi(:- q(X).)$$

e quindi è dimostrato che la computazione di σ^π non termina perché viene richiesto di soddisfare ciclicamente il goal $:- q(X).$ indefinitamente.

6 Il Linguaggio Prolog_!

Il linguaggio Prolog₀⁺, e quindi il linguaggio Prolog₀, è sufficiente per realizzare programmi che possono essere utilizzati per risolvere problemi interessanti. Ad esempio, Prolog₀⁺ è stato utilizzato per realizzare un programma per descrivere un grafo diretto aciclico che è poi stato usato per soddisfare vari goal interessanti. Però, Prolog₀⁺ non è ancora sufficiente perché non offre alcun modo per bloccare le scelte non deterministiche. Quindi, un programma Prolog₀⁺ è costretto ad esplorare tutte le scelte non deterministiche disponibili anche se non è necessario farlo oppure se per farlo è necessario attivare una computazione che non termina. Questa necessità è spesso considerata troppo vincolante perché, alle volte,

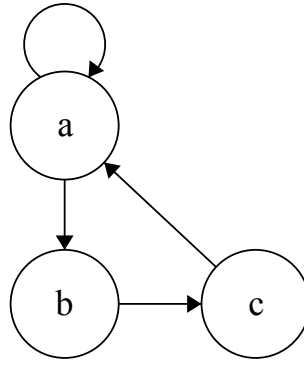


Figura 2: Un semplice grafo diretto con un autoanello e un ciclo.

è necessario poter controllare in modo dettagliato quali scelte non deterministiche attivare e quando attivarle, eventualmente facendo queste scelte in modo dinamico e dipendente dallo stato della computazione.

Si consideri, ad esempio, il grafo diretto (ciclico) riportato in Figura 2. Il grafo può essere descritto in un programma Prolog_0^+ utilizzando due predicati, come segue:

```

arc(a, a).
arc(a, b).
arc(b, c).
arc(c, a).

```

```

path(X, X, [X]).
path(X, Y, [X | R]) :- arc(X, Z), path(Z, Y, R).

```

dove le regole che descrivono `path` sono state introdotte per calcolare il percorso tra due nodi. Quindi, il seguente goal può essere utilizzato per elencare i percorsi nel grafo tra il nodo `a` e il nodo `c`:

```

path(a, c, P).

```

Però, la soluzione proposta presenta due problemi:

1. La presenza dell'autoanello sul nodo `a` innesca immediatamente un ciclo infinito che non permette di ottenere alcuna soluzione al goal; e
2. Anche dopo aver rimosso l'autoanello, ad esempio rimuovendo o commentando il primo fatto, il goal viene soddisfatto infinite volte nonostante i percorsi prodotti siano solo ripetizioni cicliche dello stesso percorso `[a, b, c]`.

Entrambi questi problemi potrebbero essere risolti richiedendo che un percorso non possa contenere due volte lo stesso nodo. Quindi, in presenza di un ipotetico operatore di negazione, sarebbe sufficiente verificare che un nodo non sia presente nel percorso corrente e, solo in questo caso, aggiungerlo.

Però, la negazione è uno strumento complesso da realizzare nell'ambito della programmazione logica soprattutto a causa della presenza di **variabili libere**, che sono variabili non ancora sostituite da termini durante la computazione. Quindi, anziché estendere Prolog_0^+ con una negazione complessa da realizzare e da utilizzare, è più utile estendere Prolog_0^+ con uno strumento in grado di inibire alcuni rami della computazione non deterministica.

Il linguaggio Prolog_1 è il linguaggio Prolog_0 arricchito di un atomo che può essere utilizzato come un congiunto e, quindi, nei goal e nei corpi delle regole. Questo operatore viene indicato con il simbolo **!** (letto **cut** o **simbolo di cut**) e viene usato solo con arità zero. Quindi, la produzione che descrive i congiunti in Prolog_1 diventa:

$$\text{Conjunct} \rightarrow \text{Atom} \mid \text{Atom}(\text{Arguments}) \mid ! \quad (19)$$

Lo strumento del cut viene spesso criticato perché è in grado di introdurre dei comportamenti inattesi se non viene usato con estrema cautela. Quindi, conviene subito sottolineare il fatto che l'uso del cut deve essere limitato allo stretto necessario.

Informalmente, quando viene incontrato un cut nel corpo di una regola $h :- b$, vengono scartate:

1. Tutte le scelte non deterministiche attivate da quando l'esecutore ha iniziato a soddisfare la regola e, quindi, tutte le scelte non deterministiche legate ai congiunti di b che precedono il cut; e
2. Tutte le scelte non deterministiche attivate per cercare un fatto o una regola con testa che unifichi h e, quindi, tutte le scelte non deterministiche legate ai fatti e alle regole che seguono la regola corrente nel testo del programma e che hanno teste che unificano con h .

Si noti che queste scelte non deterministiche vengono scartate anche se la regola in cui è presente il cut fallisce. Quindi, contrariamente agli altri effetti legati al soddisfacimento dei goal, gli effetti di un cut non vengono inibiti da un fallimento.

Ad esempio, il cut presente nel seguente programma:

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

q(y).
q(z).

r(y).
r(z).
```

garantisce che, una volta incontrato, vengano scartate le scelte non deterministiche legate a $q(X)$, $p(b)$ e $p(c)$. Quindi, il goal $:- p(W)$ viene soddisfatto unicamente dalle sostituzioni $W=a$ e $W=y$ perché $q(z)$, $p(b)$ e $p(c)$ vengono inibite dal cut.

In modo del tutto analogo, quando viene incontrato un cut in un goal, vengono scartate tutte le scelte non deterministiche attivate da quando l'esecutore ha iniziato a soddisfare il goal e, quindi, tutte le scelte non deterministiche legate ai congiunti del goal che precedono il cut. Quindi, il goal $:- p(W), !$ applicato al programma precedente viene soddisfatto unicamente dalla sostituzione $W=a$ perché la sostituzione $W=y$ viene inibita dal cut.

Normalmente, l'introduzione del cut viene accompagnata dai seguenti fatti e regole che vengono aggiunte al prologo utilizzato implicitamente con i programmi scritti in Prolog_1^+ :

```
true.

fail :- a = b.
```

Il primo fatto introduce **true**, che serve per esprimere dei goal che non falliscono mai. La seconda regola serve per introdurre **fail**, che è un goal che fallisce sempre. Usando **fail** è possibile definire in modo semplice il seguente predicato:

```
X \= X :- !, fail.
X \= Y.
```

Questo è un predicato binario, espresso mediante un operatore infisso, che serve per definire cosa significhi diverso in Prolog_I^+ dicendo che due termini sono diversi tra loro solo se non possono essere unificati. Quindi, oltre a variabili dummy, liste e operatori, Prolog_I^+ aggiunge a Prolog_I gli operatori **=** e **\=**, il fatto **true** e le regole per definire **fail**.

L'introduzione del cut permette di risolvere i problemi dell'esempio discusso in precedenza e relativo ad un grafo diretto non aciclico. In particolare, con la seguente soluzione è possibile impedire che un percorso contenga più volte lo stesso nodo:

```
node(a).
node(b).
node(c).

arc(a, a).
arc(a, b).
arc(b, c).
arc(c, a).

path(X, Y, P) :-
    path1(X, Y, [], P).

path1(_, X, V, _) :-
    member(X, V),
    !,
    fail.
path1(X, X, V, [X | V]).
path1(Y, X, V, R) :-
    arc(Z, X),
    path1(Y, Z, [X | V], R).
```

L'introduzione del cut in Prolog_I e, equivalentemente nel linguaggio esteso Prolog_I^+ , permette di modificare la computazione svolta dall'esecutore e, quindi, in generale, cambia l'insieme delle sostituzioni che risolvono un goal. Però, alle volte, l'utilizzo del cut non ha effetto sull'insieme delle sostituzioni che soddisfano un goal, ma si limita, eventualmente, solo a cambiare l'ordine con cui le soluzioni al goal vengono calcolate o il numero di volte in cui una soluzione viene prodotta. Dato un programma e un goal, se il cut viene utilizzato per modificare il programma o il goal senza però modificare l'insieme delle soluzioni al goal, allora viene detto *green cut*. Viceversa, in tutti gli altri casi, si parla di *red cut*. I green cut vengono utilizzati solo per migliorare le prestazioni dell'esecutore nel risolvere alcuni goal e, quindi, non cambiano in modo sostanziale la semantica del programma e del goal. Viceversa, i *red cut* vengono utilizzati per ottenere programmi o goal che si comportino in modo sostanzialmente diverso da quelli che non usano il cut. Normalmente, i green cut vengono considerati utili e non particolarmente pericolosi dal punto di vista della correttezza dei programmi realizzati. Viceversa, i red cut sono considerati uno strumento che andrebbe utilizzato poco e con estrema cautela.

L'introduzione del cut permette di spiegare perché la negazione è ritenuta problematica nella programmazione logica. Si consideri, ad esempio, il problema di stabilire se un elemento non è membro di una lista. Una possibile soluzione scritta in Prolog⁺ è la seguente:

```
nonmember(_, []).
nonmember(X, [X | _]) :- !, fail.
nonmember(X, [_ | R]) :- nonmember(X, R).
```

Però, questa soluzione, come tutte le soluzioni che coinvolgono la negazione, ha dei comportamenti anomali in presenza di variabili libere. Ad esempio:

```
:- nonmember(R, [a, b, c]).
false

:- nonmember(z, [a, b, R]).
false
```

Si noti che non è possibile ovviare a questi problemi anche utilizzando il linguaggio Prolog. Infatti, la negazione offerta da Prolog viene detta **negazione per fallimento** e viene realizzata soddisfacendo un goal negato, per fallimento, solo quando il relativo goal affermato non è soddisfacibile. Quindi, l'utilizzo opportuno del cut consente sempre di evitare l'utilizzo della negazione per fallimento ed è preferibile perché le situazioni anomale introdotte dal cut risultano più evidenti.

La semantica di Prolog_i può essere introdotta formalmente andando ad estendere la semantica di Prolog₀. L'estensione richiede di definire la funzione non deterministica σ^π che definisce la semantica di Prolog_i estendendo quella di Prolog₀ per produrre come risultato una coppia (θ, γ) in cui θ è la sostituzione che rappresenta il risultato della computazione mentre γ viene utilizzata per gestire il cut. In particolare, γ può valere $!$ o \perp e, se vale, $!$ significa che la computazione è stata alterata dall'utilizzo di un cut, mentre se vale \perp significa che la computazione non ha coinvolto alcun cut. Naturalmente, questo richiede di modificare anche la funzione σ_G che definisce il comportamento dell'esecutore durante il soddisfacimento dei goal. Questa funzione estende quella di Prolog₀ prevedendo la presenza di un nuovo tipo di congiunto. Questo nuovo tipo di congiunto, il cut, non fallisce mai, non produce una sostituzione ma solo un valore per γ e la funzione è fatta in modo che gli effetti della presenza di un cut si propaghino anche se i goal che seguono il cut falliscono. Quest'ultima parte è particolarmente insidiosa e complica la funzione in modo sensibile, ma è necessaria per permettere di realizzare, ad esempio, i predicati $\backslash=$ e **nonmember**. Infatti, entrambi questi predicati utilizzano il goal **!**, **fail** per garantire che non venga preso un percorso non deterministico in cui viene richiesto che i goal falliscano. Lo stesso comportamento viene tenuto dal predicato **path** dell'esempio precedente per garantire che, se un nodo è già presente in un percorso, la costruzione del percorso si interrompa.

La funzione non deterministica σ^π che definisce la semantica di Prolog_i estendendo quella di Prolog₀ produce una coppia (θ, γ) in cui θ è la sostituzione che rappresenta il risultato della computazione mentre γ viene utilizzata per gestire il cut. In particolare, γ può valere $!$ o \perp e, se vale, $!$ significa che la computazione è stata alterata dall'utilizzo del cut, mentre se vale \perp significa che la computazione non ha coinvolto il cut. Dati due valori γ_1 e γ_2 che seguono le convenzioni precedenti, viene definito:

$$\gamma_1 \oplus \gamma_2 = \begin{cases} ! & \text{se } \gamma_1 = ! \vee \gamma_2 = ! \\ \perp & \text{altrimenti} \end{cases} \quad (20)$$

In più, per definire σ^π , anche le funzioni non deterministiche σ_G^π , σ_F e σ_R^π vengono modificate in modo che producano una coppia (θ, γ) con le stesse convenzioni adottate per σ^π . Infine, anche le funzioni $head_G$ e $rest_G$ vengono modificate in modo che il cut possa essere presente in un goal.

In particolare, dato un programma π scritto in Prolog_! e un goal ν , la funzione σ^π è definita come segue:

$$\sigma^\pi(\nu) = \begin{cases} \sigma_!^\pi(g, \pi) & \text{se } g = head_G(\nu) \wedge \perp = rest_G(\nu) \\ (\theta_g \circ \theta_r, \gamma_g \oplus \gamma_r) & \text{se } g = head_G(\nu) \wedge r = rest_G(\nu) \wedge r \neq \perp \wedge \\ & (\theta_g, \gamma_g) = \sigma_!^\pi(g, \pi) \wedge (\theta_r, \gamma_r) = \sigma^\pi(\text{:- } r\theta_g.) \end{cases} \quad (21)$$

in cui tutte le scelte non deterministiche vengono scartate se $\gamma_g \oplus \gamma_r = !$ nel secondo caso. Questa possibilità permette di scartare tutte le scelte non deterministiche attivate per soddisfare un goal nel caso in cui il congiunto corrente o i congiunti successivi a quello corrente contengano un cut.

La funzione (deterministica) $\sigma_!^\pi$ riceve come primo argomento un congiunto e come secondo argomento un programma, cioè una sequenza di clausole definite, ed è definita come segue:

$$\sigma_!^\pi(g, \pi) = \begin{cases} (\emptyset, !) & \text{se } g = ! \\ \sigma_G^\pi(g, \pi) & \text{altrimenti} \end{cases} \quad (22)$$

La funzione σ_G^π riceve come primo argomento un congiunto e come secondo argomento un programma, cioè una sequenza di clausole definite. Se effettivamente applicabile agli argomenti e nell'ipotesi che termini, σ_G^π produce in modo non deterministico delle coppie (θ, γ) utilizzando le convenzioni adottate per σ^π . In particolare, la funzione σ_G^π è definita come segue:

$$\sigma_G^\pi(g, \pi) = \begin{cases} \sigma_F(g, c) & \text{se } c = head_P(\pi) \wedge c = h. \\ (\theta_r, \gamma_r) & \text{se } c = head_P(\pi) \wedge c = h \text{ :- } b. \wedge (\theta_r, \gamma_r) = \sigma_R^\pi(g, c) \\ \sigma_G^\pi(g, r) & \text{se } r = rest_P(\pi) \wedge r \neq \perp \end{cases} \quad (23)$$

in cui tutte le scelte non deterministiche vengono scartate se viene scelto il primo caso o se $\gamma_r = !$ nel terzo caso. Questa possibilità consente di scartare tutte le scelte non deterministiche che potrebbero essere attivate proseguendo con il resto del programma nel tentativo di soddisfare il goal.

La funzione σ_F riceve come primo argomento un congiunto di un goal e come secondo argomento un fatto ed è definita come segue:

$$\sigma_F(g, f) = (\theta, \perp) \quad \text{se } f' = h'. \wedge \theta = mgu(\{g, h'\}) \wedge \theta \neq \perp \quad (24)$$

Quindi, se è possibile unificare la testa del fatto con il goal, allora il MGU ottenuto dal goal e dalla testa di una variante fresh del fatto è il risultato cercato. Si noti che la funzione σ_F non coinvolge mai un cut perché un cut può essere utilizzato solo nel corpo di una regola.

In modo simile, la funzione σ_R^π riceve come primo argomento un congiunto di un goal e come secondo argomento una regola ed è definita come segue:

$$\sigma_R^\pi(g, r) = (\theta \circ \theta_r, \gamma_r) \quad \text{se } r' = h' \text{ :- } b'. \wedge \theta = mgu(\{g, h'\}) \wedge \theta \neq \perp \wedge \quad (25)$$

$$(\theta_r, \gamma_r) = \sigma^\pi(\text{:- } b'\theta.) \quad (26)$$

Si noti che la funzione σ_R^π non scarta alcuna scelta non deterministica a causa della presenza di un cut nel corpo della regola.

Esempio 6. Dato il programma $\pi = p(a). p(b).$, si vuole determinare per quali sostituzioni il goal $\nu = :- p(X), !$ è soddisfatto. In particolare,

$$\sigma^\pi(\nu) \stackrel{1}{=} \times \quad \text{non applicabile} \quad (27)$$

$$\sigma^\pi(\nu) \stackrel{2}{=} (\theta_g \circ \theta_r, \gamma_g \oplus \gamma_r) \quad (28)$$

dove:

$$(\theta_g, \gamma_r) = \sigma_!^\pi(p(X), \pi) \stackrel{1}{=} \times \quad \text{non applicabile} \quad (29)$$

$$(\theta_g, \gamma_r) = \sigma_!^\pi(p(X), \pi) \stackrel{2}{=} \sigma_G^\pi(p(X), \pi) \stackrel{1}{=} \times \quad \text{non applicabile} \quad (30)$$

$$(\theta_g, \gamma_r) = \sigma_G^\pi(p(X), \pi) \stackrel{2}{=} \sigma_F(p(X), p(a).) = (\{a/X\}, \perp) \quad (31)$$

e

$$(\theta_r, \gamma_r) = \sigma^\pi(:- !.) \stackrel{1}{=} \sigma_!^\pi(:- !.) \stackrel{1}{=} \sigma_G(!, \pi) \stackrel{1}{=} (\emptyset, !)$$

quindi, una soluzione al goal ν è $(\{a/X\}, !)$. Questa è anche l'unica soluzione perché tutte le altre scelte non deterministiche di

$$\sigma^\pi(\nu) \stackrel{2}{=} (\theta_g \circ \theta_r, \gamma_g \oplus \gamma_r) = (\{a/X\}, !)$$

vengono scartate perché $\gamma_g \oplus \gamma_r = !$.

7 Programmazione Logica con Vincoli

La programmazione logica realizzata mediante Prolog₀ e le sue varianti ed estensioni può essere ulteriormente estesa per essere resa più efficace nell'affrontare problemi comuni quali, ad esempio, i problemi di calcolo. La **programmazione logica con vincoli (Constraint Logic Programming, CLP)** estende la programmazione logica vista finora in modo molto generale e applicabile a vari contesti. Quindi, conviene subito notare che la programmazione logica con vincoli non viene introdotta unicamente per trattare problemi di calcolo mediante la programmazione logica. Anzi, è ormai così comune che quando si parla di programmazione logica, normalmente, si intende programmazione logica con vincoli.

Per estendere Prolog₀ e renderlo un linguaggio di programmazione logica con vincoli è necessario introdurre i linguaggi di vincoli. Un **linguaggio di vincoli** è una ennupla:

$$\langle D, C, A, V, sat, post, label \rangle \quad (34)$$

dove:

1. $D \neq \emptyset$ è il **dominio** del linguaggio di vincoli;
2. $C \neq \emptyset$ è un insieme di **simboli di vincolo**;
3. $A \neq \emptyset$ e $V \neq \emptyset$, con $A \cap V = \emptyset$ sono, rispettivamente, un insieme di atomi e un insieme di variabili utilizzati per costruire termini; e
4. sat , $post$ e $label$ sono tre funzioni descritte nel seguito.

L'insieme D può essere utilizzato per costruire dei CSP che abbiano come variabili gli elementi di un sottoinsieme finito di V , ammettendo anche il caso degenero in cui il sottoinsieme scelto sia vuoto. Quindi, dato un insieme finito di variabili $V_P \subseteq V$ è possibile costruire un CSP $\langle V_P, \{D\}, C_P \rangle$ che abbia V_P come insieme di variabili, D come dominio di tutte le variabili in V_P e C_P come vincoli.

Dato uno di questi CSP, la funzione *sat* dello specifico linguaggio di vincoli è in grado di verificare la soddisfacibilità del CSP. In particolare, il risultato dell'applicazione della funzione *sat* ad un CSP è \perp se la funzione è in grado di stabilire che il CSP è insoddisfacibile. Viceversa, se la funzione *sat* non è in grado di garantire l'insoddisfacibilità, allora il risultato è \top . Quindi, se il risultato della funzione *sat* applicata ad un CSP è \perp , allora il CSP è sicuramente insoddisfacibile. Viceversa, se il risultato è \top , allora il CSP potrebbe ancora essere soddisfacibile.

La funzione *post* serve per manipolare i CSP che è possibile costruire con il linguaggio di vincoli a disposizione. Infatti, dato un CSP $\langle V_P, \{D\}, C_P \rangle$, vale:

$$post(\langle V_P, \{D\}, C_P \rangle, c(t_1, t_2, \dots, t_n)) = \langle V_P \cup vars(t_1, t_2, \dots, t_n), \{D\}, C_P \cup \{\gamma\} \rangle$$

dove:

1. $c \in C$ è un simbolo di vincolo a cui è attribuita arità $n \in \mathbb{N}_+$;
2. I termini in $\{t_i\}_{i=1}^n$ sono costruiti utilizzando A e V ; e
3. γ è un vincolo che viene aggiunto al CSP.

In sintesi, la funzione *post* riceve come argomento un CSP e un'espressione che denota un vincolo γ e aggiunge il vincolo, insieme alle sue variabili, al CSP. Normalmente, la funzione *post* propaga il nuovo vincolo in modo da ridurre, se possibile, i valori ammissibili per le variabili. Comunque, non viene richiesto che la *post* propaghi i vincoli e, anche in caso lo facesse, non è previsto che la propagazione identifichi se il nuovo CSP prodotto è insoddisfacibile, anche perché non è previsto che la funzione *post* studi la soddisfacibilità dei CSP che estende. Quindi, partendo da un CSP improprio, perché senza variabili e senza vincoli, è possibile costruire nuovi CSP mediante la funzione *post*. Si noti che *post* non è definita per tutte le espressioni che può ricevere come secondo argomento. Infatti, la funzione *post* dello specifico linguaggio di vincoli a disposizione identifica anche quali sono le espressioni per le quali è possibile costruire nuovi CSP aggiungendo vincoli a CSP disponibili.

Infine, la funzione *label* ha un comportamento simile alla funzione *post*, ma è non deterministica. In particolare, dato un CSP $\langle V_P, \{D\}, C_P \rangle$, vale:

$$label(\langle V_P, \{D\}, C_P \rangle, x) = \begin{cases} (\langle V_P, \{D\}, C_P \cup \{\gamma\} \rangle, t) & \text{se il CSP ottenuto è} \\ & \text{soddisfacibile per } t \\ \perp & \text{altrimenti} \end{cases} \quad (35)$$

dove:

1. $x \in V_P$ è una variabile del problema;
2. t è un termine costruibile utilizzando A e V tale che esista un unico $d \in D$ associato a t ; e
3. γ è un nuovo vincolo che impone che la variabile x valga $d \in D$, con d valore univocamente associato a t .

Quindi, la funzione *label* riceve come primo argomento un CSP e come secondo argomento una variabile del CSP. La funzione costruisce un nuovo CSP e, se è soddisfacibile, lo ritorna insieme ad un termine. In particolare, il nuovo CSP impone che la variabile valga $d \in D$, dove d è univocamente identificato dal termine t che viene ritornato insieme al CSP.

Si noti che alcuni linguaggi di vincoli forniscono una funzione *label* in grado di verificare la soddisfacibilità del nuovo CSP ottenuto almeno per un valore d , univocamente associato a t , se esiste. Quindi, contrariamente a quanto accade utilizzando *sat*, la funzione *label* di questi linguaggi consente di verificare la soddisfacibilità di un CSP andando ad assegnare un valore alla variabile x che renda il CSP soddisfacibile, se almeno un valore con questa proprietà esiste. Comunque, anche se la funzione *label* non ha questa proprietà, è previsto che la funzione *label* possa sempre stabilire la soddisfacibilità del CSP su cui lavora nel momento in cui viene utilizzata per assegnare l'ultima variabile ancora libera.

Si noti che la funzione *label* è non deterministica perché utilizza in modo non deterministico tutti i valori d , ed i corrispondenti termini t , in grado di rendere soddisfacibile il CSP ottenuto. Viceversa, per tutti i valori $\tilde{d} \in D$ per cui il corrispondente vincolo γ rende insoddisfacibile il CSP vengono scartati.

Esempio 7. Si consideri un linguaggio di vincoli con $D = \mathbb{Z}$ che metta a disposizione un termine numerico per ogni elemento di D . Sia c il vincolo definito dalla proprietà:

$$x^2 - 4 \geq 0 \quad (36)$$

Se $V_P = \{x\}$ è un sottoinsieme dell'insieme delle variabili del linguaggio di vincoli considerato, allora vale:

$$label(\langle \{x\}, \{\mathbb{Z}\}, \{c\}, x) = (P_L, t) \quad (37)$$

dove t vale, in modo non deterministico, tutti i termini numerici univocamente associati ai valori nell'insieme:

$$\{z \in \mathbb{Z} : |z| \geq 2\} = \{\pm 2, \pm 3, \dots\} \quad (38)$$

Quindi, se applicata più volte, la funzione *label* avrà come risultato t un termine tipo $2, -2, 3, -3, \dots$ ottenendo termini diversi per ogni applicazione.

Dato un linguaggio di vincoli $\mathcal{D} = \langle D, C, A, V, sat, post, label \rangle$ è possibile estendere Prolog₀ al linguaggio CLP₀(\mathcal{D}) ottenuto permettendo di utilizzare i vincoli come se fossero dei predicati. In particolare, dal punto di vista sintattico:

1. L'insieme degli atomi di CLP₀(\mathcal{D}) contiene gli atomi di Prolog₀, i simboli di vincolo in C , gli atomi in A e la parola *label*; e
2. L'insieme delle variabili di CLP₀(\mathcal{D}) contiene tutte le variabili di Prolog₀ e tutte le variabili in V .

Si noti che non è richiesto che gli insiemi che arricchiscono CLP₀(\mathcal{D}) rispetto a Prolog₀ siano disgiunti dai rispettivi insiemi di Prolog₀. Quindi, normalmente, il linguaggio CLP₀(\mathcal{D}) ha tipicamente lo stesso insieme di variabili di Prolog₀.

Dal punto di vista semantico, le funzioni che calcolano il valore semantico di un programma scritto in CLP₀(\mathcal{D}) per un particolare goal vengono arricchite di un nuovo argomento, che è un CSP costruito mediante il linguaggio di vincoli \mathcal{D} . Oltre a questo nuovo argomento, le funzioni hanno anche un nuovo risultato, che è anch'esso un CSP costruito mediante il linguaggio di vincoli \mathcal{D} . Infatti, dato un programma e un goal scritti in

$CLP_0(\mathcal{D})$, il valore semantico calcolato è una coppia in cui la prima componente è una sostituzione e la seconda componente è un CSP che esprime i vincoli accumulati durante la computazione e relativi, almeno, alle variabili della sostituzione.

In particolare, la funzione σ_G è estesa in modo che tratti i congiunti che hanno come testa un simbolo di vincolo in modo specifico. Anziché cercare calcolare il valore semantico di un congiunto di questo tipo facendo ricorso all'unificazione, come avviene in Prolog₀, vengono utilizzate le funzioni *sat* e *post*. Quindi, dato un congiunto che abbia come testa un simbolo di vincolo, viene applicata *post* al congiunto e al CSP che viene ricevuto come nuovo argomento di σ_G . Se la funzione *post* è effettivamente definita per il congiunto, allora viene verificata la soddisfacibilità del risultato della *post* mediante *sat*. La funzione σ_G produce un risultato, in questo caso deterministico, solo nel caso in cui la funzione *sat* indichi che non è stata in grado di determinare che il CSP ottenuto dalla *post* è insoddisfacibile. In questo caso, il risultato prodotto dalla σ_G è la sostituzione vuota insieme al nuovo CSP.

Si noti che, siccome la funzione *sat* non è sempre in grado di determinare se un CSP è insoddisfacibile, $CLP_0(\mathcal{D})$ può produrre risultati anche per CSP insoddisfacibili. Però, è possibile garantire uno studio completo della soddisfacibilità mediante la funzione *label* che associa un valore alla variabile che riceve come argomento solo quando il CSP è soddisfacibile, nel caso pessimo al momento dell'assegnazione dell'ultima variabile libera. Infatti, $CLP_0(\mathcal{D})$ include nella propria semantica la possibilità di utilizzare la funzione *label* mediante un predicato che porta lo stesso nome e che può essere applicato ad una lista di variabili che vengono assegnate nell'ordine in cui sono presenti nella lista.

Infine, si noti che la semantica di $CLP_0(\mathcal{D})$ è coerente con l'unificazione. In particolare, se il linguaggio di vincoli \mathcal{D} mette a disposizione un vincolo per imporre l'uguaglianza tra due valori, allora il vincolo è soddisfatto se e soltanto se le i termini corrispondenti ai due valori sono unificabili.

Un linguaggio di vincoli molto usato è *FD* (da *Finite Domains*), che è un linguaggio di vincoli in cui:

1. Il dominio D è composto da tutti i sottoinsiemi finiti di \mathbb{Z} , da cui il nome;
2. Tra gli atomi sono presenti i simboli univocamente associati ai numeri interi e, quindi, 1, 2, ...;
3. Tra gli atomi sono presenti i simboli univocamente associati alle operazioni aritmetiche tra interi e, quindi, +, - e altri;
4. Tra i simboli di vincolo sono presenti i simboli $\# =$, $\# \setminus =$, $\# <$, $\# = <$ e altri, che permettono di imporre i comuni vincoli di confronto tra numeri interi e vengono di solito utilizzati come operatori infissi;
5. Tra i simboli di vincolo è presente *in*, tipicamente usato come operatore infisso, che permette di attribuire un dominio ad una variabile esprimendo il dominio come un'unione di intervalli espressi usando \dots e \setminus , rispettivamente, per denotare gli intervalli e farne l'unione;
6. Tra i simboli di vincolo è presente il simbolo *ins*, tipicamente usato come operatore infisso, che permette di attribuire un dominio, esattamente come *in*, però ad un insieme di variabili elencate in una lista; e
7. Tra i simboli di vincolo sono presenti simboli per esprimere vincoli di utilizzo comune, tra i quali il simbolo *all_distinct* che permette di imporre che la lista di variabili utilizzate come argomento sia formata unicamente da variabili associate a termini tra loro tutti diversi.

Il linguaggio $CLP_0(FD)$ estende $Prolog_0$ mediante i vincoli del linguaggio di vincoli FD . In modo analogo, il linguaggio $CLP(FD)$ estende $Prolog$ mediante FD . Per utilizzare SWI-Prolog come $CLP(FD)$ è sufficiente includere il relativo modulo mediante il goal

```
:- use_module(library(clpfd)).
```

che viene tipicamente posizionato all'inizio del programma. L'utilizzo di $CLP(FD)$ permette di scrivere un predicato che sia soddisfatto da una lista e dalla relativa lunghezza:

```
length([], 0).
length([_ | L], Length) :-
    Length #>= 1,
    N1 #= Length - 1,
    length(L, N1).
```

Si noti che SWI-Prolog non permette di ridefinire questo predicato perché è uno dei predicati *built-in* di basso livello del linguaggio.

Esempio 8. Il seguente programma $CLP(FD)$ permette di enumerare tutte le soluzioni del problema di criptoaritmetica $send + more = money$:

```
:- use_module(library(clpfd)).

send(Vars) :-
    Vars = [S,E,N,D,M,O,R,Y],
    Vars ins 0..9,
    M #\= 0, S #\= 0,
    all_distinct(Vars),
    S*1000 + E*100 + N*10 + D +
    M*1000 + O*100 + R*10 + E #=
    M*10000 + O*1000 + N*100 + E*10 + Y,
    label(Vars).
```

Un altro linguaggio di vincoli interessante è *Dif*. Questo linguaggio offre un solo vincolo *dif* che permette di imporre che il termine a primo argomento sia sempre diverso dal termine a secondo argomento, intendendo che due termini sono diversi se non sono unificabili. Nonostante questo linguaggio di vincoli sia minimale, è particolarmente rilevante perché $CLP(Dif)$ permette spesso di ovviare ai noti problemi dei predicati che implicano una negazione.

Esempio 9. Il seguente predicato permette di imporre che il primo argomento non sia mai incluso nella lista che viene utilizzata come secondo argomento:

```
:- use_module(library(dif)).

nevermember(_, []).
nevermember(X, [H | R]) :-
    dif(X, H),
    nevermember(X, R).
```

8 Prolog e Logica

Nonostante il linguaggio Prolog sia l'esempio più tipico di un linguaggio di programmazione logica, non è ancora stato esplorato il legame tra Prolog e la Logica. In particolare, non è ancora stato affrontato il problema di capire se i risultati prodotti da un programma scritto in Prolog possono essere considerate conseguenze logiche valide dei fatti e delle regole incluse nel programma. Per affrontare questo problema, seguendo la strada tracciata da *Robert A. Kowalski*, è però prima necessario introdurre i linguaggi della logica dei predicati e descrivere il metodo della risoluzione.

Un linguaggio della **logica dei predicati** (o della **logica del primo ordine**) è un linguaggio che può essere costruito partendo da una **signature** $\langle P, C, F, V \rangle$ dove:

1. $P \neq \emptyset$ è un insieme di **simboli di predicato** (o **predicati**), ognuno dei quali è associato ad un'arità;
2. $C \neq \emptyset$ è un insieme di **simboli di costante** (o **costanti**);
3. F è un insieme di **simboli di funzione** (o **funzioni**), ognuno dei quali è associato ad un'arità; e
4. $V \neq \emptyset$ è un insieme infinito numerabile di **simboli di variabile** (o **variabili**).

Si noti che viene richiesto che $V \cap C = \emptyset$ e che gli insiemi non contengano le parentesi tonde, il punto e alcuni simboli che vengono comunemente utilizzati nei linguaggi di questo tipo per altri scopi, quali, ad esempio, \wedge , \vee e \top .

Partendo da una signature $\langle P, C, F, V \rangle$ è possibile costruire il relativo linguaggio della logica dei predicati mediante una struttura a due livelli. Il primo livello definisce la forma dei **termini (del primo ordine)**, esattamente come descritta in precedenza parlando del linguaggio dei termini. In particolare, usando i simboli di C come atomi con arità fissata a zero, i simboli di F come atomi con arità non nulla e i simboli di V . Poi, partendo dai termini, è possibile costruire il secondo livello della struttura del linguaggio. Questo livello contiene l'insieme delle **formule ben formate** (o **formule ben formulate**) del linguaggio. Le formule ben formulate di un linguaggio di questo tipo vengono chiamate **proposizioni** e vengono descritte partendo dai termini costruiti nel primo livello del linguaggio come segue:

1. Se t_1, t_2, \dots, t_n sono termini e p è un simbolo di predicato di arità $n \in \mathbb{N}_+$, allora $p(t_1, t_2, \dots, t_n)$ è una proposizione che viene comunemente detta *letterale affermato*;
2. I simboli \top (letto *top*) e \perp (letto *bottom*) sono proposizioni;
3. Se A è una proposizione, allora $\neg(A)$ è una proposizione e se A è un letterale affermato, allora $\neg(A)$ è un *letterale negato*;
4. Se A e B sono proposizioni, allora $(A \wedge B)$, $(A \vee B)$, $(A \implies B)$, and $(A \iff B)$ sono proposizioni;
5. Se A è una proposizione e $x \in V$ è una variabile, allora $\exists x.(A)$ e $\forall x.(A)$ sono proposizioni; e
6. Nient'altro è una proposizione.

Si noti che è consentito rimuovere le parentesi tonde se una proposizione è composta da un unico predicato o dai simboli \top e \perp e quando viene adottata la seguente precedenza per gli operatori binari: $\wedge, \vee, \implies, \iff$.

I quantificatori definiscono un *campo d'azione* (o *scope*) per una variabile e, data una proposizione, ogni occorrenza di una variabile nel campo di azione di un quantificatore viene detta *occorrenza vincolata*. Viceversa, ogni occorrenza di una variabile in una proposizione che non sia nel campo d'azione di un quantificatore viene detta *occorrenza libera*. Una proposizione viene detta *chiusa* se ha unicamente variabili con occorrenze vincolate.

Dato un insieme $D \neq \emptyset$, detto **dominio del linguaggio**, e una signature $\langle P, C, F, V \rangle$ è possibile associare un **valore di verità** (o **valore semantico**) alle proposizioni del linguaggio costruito mediante la signature a disposizione scegliendo una **funzione di interpretazione** I con le seguenti caratteristiche:

1. Per ogni $c \in C$ esiste un elemento $I(c) \in D$ chiamato interpretazione del simbolo di costante c ;
2. Per ogni $f \in F$ di arità $n \in \mathbb{N}_+$ esiste una funzione $I(f) : D^n \rightarrow D$ chiamata interpretazione del simbolo di funzione f ; e
3. Per ogni $p \in P$ di arità $n \in \mathbb{N}_+$ esiste una relazione di arità n definita su D , $I(p) \subseteq D^n$, chiamata interpretazione del simbolo di predicato p .

Fissata una funzione di interpretazione e un'assegnazione $s : V \rightarrow D$ è possibile associare un'interpretazione ai termini come segue:

1. L'interpretazione di un simbolo di costante $c \in C$ (per l'assegnazione s) è $I_s(c) = I(c)$;
2. L'interpretazione di un simbolo di variabile $x \in V$ per l'assegnazione s è $I_s(x) = s(x)$;
3. L'interpretazione di un *termine strutturato* $t = f(t_1, t_2, \dots, t_n)$ per un assegnazione s è $I_s(t) = I(f)(I_s(t_1), I_s(t_2), \dots, I_s(t_n))$.

Fissata un'interpretazione I e un'assegnazione s è possibile dire quando l'interpretazione I soddisfa una proposizione A secondo l'assegnazione s , $(I, s) \models A$. Si noti subito che, a seconda dell'assegnazione considerata, una singola proposizione potrà essere soddisfatta da un'interpretazione I o meno. Per verificare se un'interpretazione I soddisfa una proposizione A secondo un'assegnazione s si utilizzano le seguenti regole, dove $s_{x \mapsto d}$ è l'assegnazione identica ad s tranne che per la variabile x che viene assegnata a d :

1. $(I, s) \models p(t_1, t_2, \dots, t_n)$ se e soltanto se $(I_s(t_1), I_s(t_2), \dots, I_s(t_n)) \in I(p)$
2. $(I, s) \models \top$ e $(I, s) \not\models \perp$;
3. $(I, s) \models \neg A$ se e soltanto se $(I, s) \not\models A$;
4. $(I, s) \models A \wedge B$ se e soltanto se $(I, s) \models A$ e $(I, s) \models B$;
5. $(I, s) \models A \vee B$ se e soltanto se $(I, s) \models A$ oppure $(I, s) \models B$;
6. $(I, s) \models A \implies B$ se e soltanto se $(I, s) \not\models A$ oppure $(I, s) \models B$;
7. $(I, s) \models A \iff B$ se e soltanto se $(I, s) \models A$ e $(I, s) \models B$, o $(I, s) \not\models A$ e $(I, s) \not\models B$;
8. $(I, s) \models \exists x.A$ se e soltanto se esiste un $d \in D$ tale che $(I, s_{x \mapsto d}) \models A$; e
9. $(I, s) \models \forall x.A$ se e soltanto se per ogni $d \in D$ vale $(I, s_{x \mapsto d}) \models A$.

Si noti che $(I, s) \models A$ viene anche letto: A è vera in I secondo l'assegnazione s . In particolare, si dirà che A è vera in I , $I \models A$, se A è vera in I per ogni sostituzione s . In questo caso, I viene detta **modello** di A . Infine, si dirà che A non è vera in I , $I \not\models A$, se non esiste alcuna sostituzione s tale per cui A è vera in I per s . In questo caso, I viene detta **contromodello** di A .

Si consideri una **teoria (del primo ordine)** T , che non è altro che un insieme di proposizioni. La teoria si dice vera in un'interpretazione I e secondo un'assegnazione s , $(I, s) \models T$, se e soltanto se per ogni proposizione P in T vale $(I, s) \models P$. Una teoria si dice vera in un'interpretazione I , $I \models T$, se la teoria è vera per ogni assegnazione. In questo caso, l'interpretazione I si dice modello della teoria. Analogamente è possibile definire quando un'interpretazione non è vera in una teoria e quindi definire i contromodelli di una teoria. Data una teoria T e una proposizione A , non necessariamente contenuta in T , si dice che A è **conseguenza logica** di T , $T \models A$, se per ogni interpretazione I e sostituzione s che rendono vera la teoria T vale anche che $(I, s) \models A$. Quindi, per ogni interpretazione e assegnazione che rendono vera la teoria anche la conseguenza logica risulta vera. Naturalmente, non viene richiesto che la conseguenza logica sia vera unicamente per le interpretazioni e le sostituzioni che rendono vera T .

Date una teoria T e una proposizione A è possibile studiare se $T \models A$ mediante il **metodo della risoluzione**, che è un particolare tipo di **metodo di refutazione** (o **metodo di riduzione all'assurdo**). I metodi di refutazione si basano sul fatto che $T \models A$ se e soltanto se non esistono un'interpretazione I e una sostituzione s tali che $(I, s) \models T \cup \{\neg A\}$.

Il metodo di risoluzione può essere applicato per studiare la **soddisfacibilità** di un insieme generico di proposizioni, quindi l'esistenza di un modello per l'insieme di proposizioni, mediante il metodo dovuto a *Albert Thoralf Skolem*. Però, nell'ambito della programmazione logica, si prevede che le teorie siano formate unicamente da proposizioni di un particolare tipo detto **clausole di Horn**, dal nome di *Alfred Horn*.

Una clausola di Horn è una proposizione in cui valgono le seguenti restrizioni che è possibile verificare a livello puramente sintattico:

1. Non sono ammessi i quantificatori esistenziali;
2. I quantificatori universali, se presenti, sono unicamente nella parte iniziale della proposizione;
3. La parte della proposizione che segue i quantificatori universali è formata unicamente da una disgiunzione di letterali; e
4. Non più di un letterale può essere positivo.

Quindi, una clausola di Horn è una proposizione del tipo:

$$\forall x_1. \forall x_2. \dots \forall x_n. (L_1 \vee L_2 \vee \dots \vee L_m) \quad (39)$$

e non più di uno dei letterali in $\{L_i\}_{i=1}^m$ è positivo. Una clausola di Horn si dice **clausola definita** se ammette un solo letterale positivo. Mentre una clausola di Horn si dice **clausola goal** se ammette zero letterali positivi. Si noti che si ammette, impropriamente, la possibilità di avere una **clausola (di Horn) vuota**, cioè con zero letterali, per indicare \perp . Infine, conviene notare che se viene tralasciato il vincolo che prevede la presenza di non più di un letterale positivo in una clausola di Horn, allora le proposizioni vengono dette semplicemente **clausole**.

Si consideri una teoria T formata unicamente da clausole di Horn, se esistono due clausole nella teoria

$$\forall x_1. \forall x_2. \dots \forall x_n. (L_1 \vee L_2 \vee \dots \vee L_m) \quad (40)$$

$$\forall y_1. \forall y_2. \dots \forall y_r. (M_1 \vee M_2 \vee \dots \vee M_s) \quad (41)$$

che non condividono variabili e tali che esistano $1 \leq i \leq n$ e $1 \leq j \leq s$ per cui $L_i \theta = \neg M_j \theta$, con $\theta = mgu(L_i, M_j)$, allora è possibile costruire la **clausola risolvente** R di L_i ed M_j :

1. Applicando la sostituzione θ ad ognuno dei letterali delle due clausole, esclusi L_i ed M_j ;
2. Unendo tutti i letterali ottenuti dall'applicazione della sostituzione θ , e quindi escludendo L_i ed M_j , mediante delle disgiunzioni; e
3. Quantificando universalmente su tutte le variabili contenute nei nuovi letterali ottenuti dall'applicazione della sostituzione.

La clausola risolvente R è stata costruita con la cosiddetta **regola di risoluzione** e si può dimostrare che la clausola risolvente R è conseguenza logica di T . Si noti che la regola di risoluzione può produrre la clausola vuota e, in questo caso, il risultato viene più propriamente indicato con \perp . In più, si noti che la clausola risolvente è ottenuta da due clausole che non condividono variabili, quindi se si considerano clausole che condividono variabili sarà sufficiente sostituire queste variabili in una delle due clausole con delle nuove variabili per poter applicare la regola. Infine, si noti che è stata applicata l'unificazione al linguaggio di termini formato da un alfabeto che contiene anche i simboli di predicato.

Data una teoria T formata unicamente da clausole definite è possibile dimostrare che una proposizione G del tipo:

$$\exists x_1. \exists x_2. \dots \exists x_n. L_1 \wedge L_2 \wedge \dots \wedge L_m \quad (42)$$

dove i letterali in $\{L_i\}_{i=1}^m$ sono tutti affermati, è conseguenza logica di T cercando di applicare iterativamente la regola di risoluzione alla teoria $T \cup \{\neg G\}$ fino ad ottenere la clausola vuota. Infatti, è vero che se $T \cup \{\neg G\} \vdash_{RES} \perp$ allora $T \cup \{\neg G\} \models \perp$ e quindi $T \models G$. Si noti che l'applicazione iterata della regola di risoluzione fa aumentare il numero di clausole contenute nella teoria perché, per ogni applicazione della regola, si estende la teoria con la clausola risolvente ottenuta.

Esempio 10. Si consideri un linguaggio della logica dei predicati con simboli di predicato in $P = \{p\}$, simboli di costante in $C = \{a, b, e\}$, simboli di funzione in $F = \{f\}$ e simboli di variabile in V che contiene tutte le parole che iniziano con una lettera maiuscola. Partendo dalla seguente teoria espressa in forma di clausole, dove, come di consueto, sono stati lasciati impliciti i quantificatori:

- (1) $p(e, X, X)$
- (2) $p(f(X, Y), Z, f(X, W)) \vee \neg p(Y, Z, W)$

si vuole dimostrare che esiste un T tale che $p(f(a, e), T, f(a, f(b, e)))$. Per farlo, è possibile procedere *in avanti* (o *forward chaining*), come segue:

$$(3) \quad p(f(X, e), S, f(X, S)) \quad RES(1, 2) \quad \{S/W, e/Y, S/Z\}$$

da cui è possibile ottenere che la sostituzione $\{f(b, e)/S, a/X\}$ applicata a (3) porta al risultato cercato applicando la sostituzione $\{f(b, e)/T\}$ alla proposizione goal che si vuole dimostrare. In alternativa è possibile procedere *all'indietro* (*backward chaining*) e quindi per refutazione. Per prima cosa è necessario negare la proposizione goal che si vuole dimostrare ottenendo la seguente clausola goal:

$$(3) \quad \neg p(f(a, e), T, f(a, f(b, e)))$$

che viene aggiunta alla teoria. A questo punto è possibile procedere come segue:

$$\begin{array}{ll} (4) & \neg p(e, U, f(b, e)) \quad RES(2, 3) \quad \{e/Y, U/Z, f(b, e)/W\} \\ (5) & \perp \quad RES(1, 4) \end{array}$$

da cui si evince, procedendo a ritroso con le sostituzioni, che la conseguenza logica della teoria iniziale si ottiene applicando la sostituzione $\{f(b, e)/T\}$ alla proposizione goal che si vuole dimostrare.

Il legame tra la programmazione in linguaggio Prolog e il metodo della risoluzione risulta evidente riscrivendo le clausole definite come:

$$\forall x_1. \forall x_2. \dots \forall x_n. (P \Leftarrow \exists y_1. \exists y_2. \dots \exists y_r. L_1 \wedge L_2 \wedge \dots \wedge L_s) \quad (43)$$

dove P è l'unico letterale affermato della clausola definita considerata, e dipende solo dalle variabili in $\{x_i\}_{i=1}^n$, e sono state sfruttate alcune **equivalenze logiche** che riguardano i quantificatori, la negazione e l'implicazione per operare la riscrittura. Infatti:

$$\forall x_1. \forall x_2. \dots \forall x_n. \forall y_1. \forall y_2. \dots \forall y_r. (P \vee \neg L_1 \vee \neg L_2 \vee \dots \vee \neg L_s) \quad (44)$$

può essere riscritta come:

$$\forall x_1. \forall x_2. \dots \forall x_n. (P \vee \forall y_1. \forall y_2. \dots \forall y_r. (\neg L_1 \vee \neg L_2 \vee \dots \vee \neg L_s)) \quad (45)$$

e quindi:

$$\forall x_1. \forall x_2. \dots \forall x_n. (P \Leftarrow \neg(\forall y_1. \forall y_2. \dots \forall y_r. (\neg L_1 \vee \neg L_2 \vee \dots \vee \neg L_s))) \quad (46)$$

da cui si ottiene:

$$\forall x_1. \forall x_2. \dots \forall x_n. (P \Leftarrow \exists y_1. \exists y_2. \dots \exists y_r. \neg(\neg L_1 \vee \neg L_2 \vee \dots \vee \neg L_s)) \quad (47)$$

e, infine, risulta:

$$\forall x_1. \forall x_2. \dots \forall x_n. (P \Leftarrow \exists y_1. \exists y_2. \dots \exists y_r. (L_1 \wedge L_2 \wedge \dots \wedge L_s)) \quad (48)$$

Quindi, un programma scritto in Prolog non è altro che una teoria logica espressa mediante clausole definite. I fatti sono clausole definite senza letterali negati e le regole sono clausole definite con almeno un letterale negato. Il goal necessario per attivare la computazione di un programma scritto in Prolog non è altro che una proposizione per la quale si vuole verificare se sia conseguenza logica della teoria. La funzione semantica che descrive la computazione svolta dal programma a fronte di un goal non è altro che l'applicazione della regola di risoluzione secondo una strategia, rigida ma efficiente, che prende il nome di **SLD** (da **Selective Linear Definite clause resolution**). Quindi, qualsiasi sostituzione ottenuta da un programma scritto in Prolog a fronte di un goal è una conseguenza logica valida dei fatti e delle regole scritte nel programma.

9 Problemi di Pianificazione

Dato un **agente** in grado di compiere **azioni** nell'**ambiente** in cui è immerso, un **problema di pianificazione** riguarda la scelta di un **piano** di azioni che l'agente potrà svolgere per portare il **mondo** (ambiente e agente) dallo **stato attuale** in uno degli stati di **goal** che l'agente cerca di raggiungere. Per semplicità, la trattazione seguente è limitata a problemi di pianificazione simili a quelli supportati dallo *Stanford Research Institute Problem Solver* (*STRIPS*), che rappresentano un particolare tipo di **problemi di pianificazione proposizionale**. In particolare, è possibile definire un problema di pianificazione come una quintupla:

$$\langle P, A, I, G, \tilde{G} \rangle \quad (49)$$

dove:

1. $P \neq \emptyset$ è un insieme finito di *proposizioni* (della logica proposizionale);
2. $A \neq \emptyset$ è un insieme finito di *azioni*;
3. $I \neq \emptyset$ è un sottoinsieme finito di P detto *stato iniziale* (del mondo);
4. $G \neq \emptyset$ è un sottoinsieme finito di P detto *goal asserito*; e
5. $\tilde{G} \neq \emptyset$ è un sottoinsieme finito di P detto *goal negato*.

In ogni istante, il mondo è caratterizzato da uno *stato* che viene descritto da un sottoinsieme di P che contiene tutte e sole le proposizioni che risultano vere nello stato. Quindi, ogni proposizione non contenuta nella descrizione di uno stato risulta falsa nello stato stesso. In particolare, I contiene tutte e sole le proposizioni ritenute vere nello stato iniziale del problema di pianificazione. Si noti, però, che G e \tilde{G} non sono descrizioni di uno stato, ma sono, rispettivamente, l'insieme delle proposizioni che devono essere vere e quelle che devono essere false in uno stato che possa essere considerato uno *stato di goal* del problema di pianificazione. Una soluzione del problema di pianificazione è una sequenza finita di azioni che, una volta eseguite dall'agente, portino il mondo dallo stato iniziale I ad uno stato di goal F che rispetti i requisiti espressi mediante G e \tilde{G} e, quindi, tale che:

$$G \subseteq F \quad \wedge \quad \tilde{G} \cap F = \emptyset \quad (50)$$

Ogni elemento di A è un'azione ed è una quintupla:

$$\langle n, R, \tilde{R}, T, \tilde{T} \rangle \quad (51)$$

dove:

1. n è un simbolo detto *nome* (o *identificativo*) dell'azione e determina univocamente ogni elemento di A ;
2. R è un sottoinsieme di P detto *precondizione asserita*;
3. \tilde{R} è un sottoinsieme di P detto *precondizione negata* tale che $R \cap \tilde{R} = \emptyset$;
4. T è un sottoinsieme di P detto *postcondizione asserita*; e
5. \tilde{T} è un sottoinsieme di P detto *postcondizione negata* tale che $T \cap \tilde{T} = \emptyset$.

Un'azione $a \in A$, con $a = \langle n, R, \tilde{R}, T, \tilde{T} \rangle$, può essere compiuta dall'agente se il mondo si trova in uno stato S tale che:

$$R \subseteq S \quad \wedge \quad \tilde{R} \cap S = \emptyset \quad (52)$$

Se l'agente compie l'azione a nello stato S , che soddisfa le precondizioni precedenti per ipotesi, allora il mondo si porta nello stato S' tale che:

$$S' = (S \setminus \tilde{T}) \cup T \quad (53)$$

Per risolvere un problema di pianificazione è possibile operare una **ricerca nello spazio degli stati** che parta dallo stato iniziale I e proceda fino all'identificazione delle soluzioni al problema, se esistono. La ricerca avviene mediante un insieme di *nodi*, ognuno dei quali contiene uno stato S , l'insieme degli stati attraversati per arrivare dallo stato iniziale fino allo stato S e la sequenza di azioni che ha permesso di arrivare fino allo stato S . Si noti che l'insieme degli stati attraversati per arrivare fino allo stato S è importante perché permette di capire se la ricerca sta per proseguire in uno stato già attraversato e, in questo caso, bloccarla in modo da garantire che la ricerca termini sempre. I nodi vengono organizzati in un **albero di ricerca** che ha come radice il nodo corrispondente allo stato I , il quale ha un insieme vuoto di stati già attraversati e viene raggiunto con una sequenza di azioni vuota. I nodi figli di un nodo N sono ottenuti andando a compiere tutte le azioni possibili partendo dal nodo N . Fissato un percorso nell'albero di ricerca, l'insieme dei nodi costruiti e non ancora scartati viene detto *frangia* (o *fringe*). In sintesi, la ricerca nello spazio degli stati può essere riassunta come segue:

1. La frangia viene inizializzata con lo stato iniziale I , un insieme di stati attraversati vuoto e la sequenza di azioni vuota;
2. Se la frangia è vuota, allora è stato provato che non esiste una soluzione al problema di pianificazione;
3. Viceversa, viene prelevato il nodo H alla testa della frangia e viene indicato con S il corrispondente stato, con L l'insieme degli stati attraversati e con P la corrispondente sequenza di azioni;
4. Se S è uno stato di goal, allora viene ritornato il piano ottenuto dalla sequenza di azioni P ;
5. Viceversa, lo stato S viene *espanso* generando l'insieme degli stati raggiungibili da S mediante l'applicazione di tutte le azioni disponibili;
6. Per ognuno degli stati S' raggiungibili da S mediante l'applicazione di un'azione a , se $S' \in L$ allora lo stato viene scartato;
7. Viceversa, viene costruito un nodo relativo ad S' usando come insieme degli stati attraversati $L \cup S$ ed estendendo la sequenza di azioni P con l'azione a , quindi il nuovo nodo viene aggiunto alla frangia; e
8. Si ritorna al punto 2.

Il metodo così sommariamente descritto ha almeno un punto in cui può essere raffinato. In particolare, non viene deciso in quale posizione della frangia verrà aggiunto un nuovo nodo ottenuto dall'espansione di un altro nodo.

Se ogni nuovo nodo viene:

- Aggiunto in fondo alla frangia, allora l'algoritmo di ricerca nello spazio degli stati viene detto **in ampiezza** (**Breadth-First Search** o **BFS**).
- Aggiunto in testa alla frangia, allora l'algoritmo di ricerca nello spazio degli stati viene detto **in profondità** (**Depth-First Search** o **DFS**).
- Aggiunto in modo che gli stati più vicini agli stati di goal siano più vicini alla testa della frangia, allora l'algoritmo di ricerca nello spazio degli stati viene detto **informato**, per distinguerlo dai precedenti due algoritmi che vengono detti **non informati**.

Si noti che tutti i metodi elencati hanno complessità computazionale temporale asintotica di caso pessimo esponenziale di ordine $O(b^d)$, dove $b = |A|$ e d è il numero di azioni necessarie per raggiungere, nel caso pessimo, uno stato di goal. Però, gli algoritmi hanno caratteristiche diverse da almeno altri due punti di vista:

1. BFS ha complessità computazionale spaziale asintotica di caso pessimo di ordine $O(b^d)$ e garantisce che, per problemi risolubili, venga ottenuta una soluzione composta dal numero minimo di azioni; ma
2. DFS ha complessità computazionale spaziale asintotica di caso pessimo di ordine $O(d)$ pur non garantendo, per problemi risolubili, che venga ottenuta una soluzione composta dal numero minimo di azioni.

Per ottenere un buon compromesso tra la BFS e la DFS, di solito si introduce la **ricerca ad approfondimenti successivi (della ricerca in profondità)** (**Iterative Deepening Depth-First Search, ID-DFS**) che lavora nel seguente modo:

1. Viene inizializzato il *parametro di profondità massima* N al valore uno;
2. Viene applicata la DFS fino alla costruzione di un albero di ricerca con non più di N livelli;
3. Se viene trovata una soluzione, allora viene immediatamente ritornata, senza necessariamente completare l'albero di ricerca; ma
4. Se non viene trovata una soluzione, allora viene incrementato N di uno e si ritorna al punto 2.

La ID-DFS ha la complessità computazionale spaziale asintotica di caso pessimo della DFS, ma garantisce che, per problemi risolubili, venga trovata una soluzione composta dal numero minimo di azioni, come accade per la BFS. Si noti che, a causa degli incrementi di N , la DFS viene applicata ad alberi che condividono la parte più vicina alla radice. Questo prevede che vengano costruiti più volte i nodi dei primi livelli degli alberi, senza però peggiorare la complessità computazionale temporale asintotica. Infatti, il numero di nodi di un albero finito è sempre di ordine $O(b^d)$, dove b è il numero massimo di figli di ogni nodo e d è la profondità dell'albero. Quindi, l'ordine del numero dei nodi di un albero di profondità finita dipende unicamente dal numero dei nodi nella frangia dell'albero.

La ricerca informata normalmente richiede che:

1. Il costo sia *definito positivo e additivo*; e
2. Venga definita una *funzione di costo* $c = f(N)$ per ogni nodo N che quantifichi il costo minimo necessario per partire dalla radice, passare per N e arrivare ad uno stato di goal.

In queste ipotesi, dato un nodo N ottenuto durante la costruzione dell'albero di ricerca, vale:

$$f(N) = h(N) + g(N) \quad (54)$$

dove $g(N)$ è il costo necessario per raggiungere il nodo N partendo da I e h è il costo necessario per raggiungere il più vicino stato di goal. Però, spesso, la funzione h non è disponibile e quindi si ricorre alla funzione $h^*(N)$ tale che:

$$0 \leq h^*(N) \leq h(N) \quad (55)$$

per ogni nodo N . La funzione h^* viene detta **funzione euristica** e se i nodi vengono inseriti nella frangia in senso crescente di $f^*(N) = h^*(N) + g(N)$, allora l'algoritmo di ricerca viene chiamato A^* . Si può dimostrare che A^* è ottimo nel senso che richiede il numero minimo di espansioni di nodi per ottenere una soluzione a costo minimo, se esiste.

Infine, si noti che la ricerca nello spazio degli stati che è stata descritta viene detta **in avanti** (o **forward chaining**) perché parte dallo stato iniziale fino a raggiungere uno stato di goal. Viceversa, è possibile modificare la ricerca in modo che parta dagli stati di goal e prosegua verso lo stato iniziale facendo, quindi, una ricerca **all'indietro** (o **backward chaining**), sostanzialmente, invertendo il ruolo di precondizioni e postcondizioni.