Rilassamento in reti di vincoli e metaprogrammazione logica

Francesca Rossi */** e Ugo Montanari *

* Dipartimento di Informatica, Università di Pisa
Corso Italia 40, 56100 Pisa

** Selenia S.p.A., Roma

Sommario

Il problema di risolvere una rete di vincoli è NP-completo. Per questa ragione, anche se l'algoritmo standard per la soluzione di reti di vincoli rimane il backtracking, sono stati sviluppati molti algoritmi di rilassamento (da applicare prima del backtracking). Lo scopo di tali algoritmi è quello di migliorare la complessità, nel caso medio, della ricerca con backtracking, che deve comunque poi essere applicata.

Questo lavoro mostra come è possibile esprimere sia le reti di vincoli che gli algoritmi di rilassamento in termini di programmi e metaprogrammi logici. Inoltre, l'uso degli algoritmi di rilassamento in programmazione logica viene poi visto come un metodo generale per la trasformazione di programmi logici che hanno la stessa forma di quelli che descrivono le reti di vincoli, o di programmi logici che possono essere semplicemente ridotti a questa forma mediante una elaborazione simbolica finita.

1. Introduzione

In questo lavoro intendiamo mostrare come è possibile utilizzare la programmazione logica per risolvere efficientemente le reti di vincoli. Inoltre, descriviamo come i metodi di rilassamento (solitamente usati nel campo delle reti di vincoli per renderle più esplicite e quindi più semplici da risolvere) possano essere visti come un metodo generale di trasformazione di programmi logici.

Le reti di vincoli sono un semplice modello dichiarativo di rappresentazione della conoscenza, utile per descrivere tutti quei problemi dove è richiesta la soddisfazione simultanea di più vincoli. In Intelligenza Artificiale sono molto importanti, in quanto, per esempio, problemi come la specifica di sistemi fisici e software, la soddisfattibilità booleana, e l'etichettamento di scene risultano naturalmente esprimibili in termini di reti di vincoli.

Qui, le reti di vincoli sono definite come ipergrafi etichettati: i nodi rappresentano variabili a cui assegnare dei valori, mentre gli iperarchi (cioè archi connessi a una, due, o più variabili) sono vincoli che devono essere soddisfatti dai valori delle variabili adiacenti; infatti, ogni vincolo connesso a k variabili è etichettato da una relazione che specifica le tuple di valori di tali k variabili che soddisfano il vincolo. Una soluzione di una rete di vincoli è definita qui come un assegnamento di valori ad alcune tra le variabili della rete che soddisfa simultaneamente tutti i vincoli. La selezione di tali variabili è visibile nella stessa definizione della rete tramite l'indicazione di un iperarco che le connette.

Sfortunatamente, il problema di risolvere una rete di vincoli è NP-completo. Quindi, qualsiasi algoritmo che risolve una generica rete avrà, nel caso pessimo, una complessità esponenziale nel numero delle variabili della rete.

Vediamo ad esempio il comportamento di due algoritmi di soluzione molto conosciuti.

L'istanziazione di tutte le variabili in tutti i modi possibili per poi mantenere solo quelle istanziazioni che soddisfano tutti i vincoli (algoritmo di "generate and test") risulta sempre di complessità esponenziale (sia nel caso migliore che nella media che nel caso pessimo).

L'algoritmo di backtraking, invece, che è quello usato solitamente per risolvere le reti, è lineare nel caso migliore ed esponenziale nel caso pessimo, e può quindi essere accettabile in molti casi. Ma il backtracking, pur essendo un miglioramento rispetto all'algoritmo di generate and test, può produrre un comportamento detto di "thrashing": può cioè ripetutamente istanziare la stessa variabile allo stesso valore un grande numero di volte prima di scoprire che tale istanziazione non partecipa a nessuna soluzione.

Proprio per diminuire la possibilità di tale comportamento sono state sviluppate delle tecniche di consistenza, che cercano di scoprire un'inconsistenza il più presto possibile e possono essere applicate alle reti sia prima che durante il backtracking.

Le tecniche di consistenza da usare durante il backtracking controllano, ogni volta che una nuova variabile viene istanziata, la consistenza di un piccolo insieme di vincoli, scelti in relazione a tale variabile. Esempi di queste tecniche sono il forward checking, il look ahead e il partial look ahead.

Le tecniche di consistenza da usare invece prima del backtracking, chiamate anche algoritmi di rilassamento, sono basate sull'idea che, più i

vincoli della rete da risolvere sono espliciti (cioè riflettono il vincolo globale indotto dall'intera rete su di essi), e più il backtracking è efficiente. Essi quindi prendono una rete e ne restituiscono un'altra più esplicita.

In questo lavoro un algoritmo di rilassamento è definito come l'applicazione di alcune regole di rilassamento in una certa sequenza, finchè la rete non viene più modificata (a questo punto abbiamo la rete stabile, che è detta chiusura della rete data rispetto alle regole applicate). Ogni regola di rilassamento cambia la definizione di un vincolo della rete, restringendola al fine di tener conto dei vincoli di una sottorete adiacente. L'insieme di regole usate da un algoritmo di rilassamento è detto adeguato se la rete stabile è risolta (cioè il vincolo che connette le variabili selezionate è definito proprio dall'insieme di tutte le soluzioni della rete), ed è detto perfetto se è adeguato ed in più la rete stabile è stata ottenuta applicando ogni regola una sola volta.

Esempi di algoritmi di rilassamento sono quelli che ottengono la consistenza a livello di arco e a livello di cammino (*arc-consistency* e *path-consistency*) ([7,10]), l'algoritmo di *synthesis* di Freuder ([4,5]), e alcune loro varianti come gli algoritmi di *directional arc-consistency*, *directional path-consistency* e *adaptive consistency* sviluppati da Decther e Pearl in [3].

In questo lavoro mostriamo come usare la programmazione logica per risolvere le reti di vincoli, sia usando il backtracking che gli algoritmi di rilassamento.

Per prima cosa, diamo una corrispondenza sintattica e semantica tra le reti di vincoli e particolari programmi logici detti *network programs*: data una rete di vincoli, è semplice scrivere un corrispondente *network program* (ha una sola regola che descrive la rete, più un insieme di fatti che definiscono i vincoli), la cui semantica standard coincide con la soluzione esatta della rete (ogni soluzione è ottenuta usando proprio il backtracking, che viene normalmente usato dall'interprete standard del linguaggio Prolog).

Inoltre, dato un algoritmo di rilassamento (cioè un insieme di regole di rilassamento e l'ordine in cui applicarle), possiamo scrivere un programma logico corrispondente, detto *relaxation program*.

Ora, se eseguiamo insieme un *network program* e un *relaxation program*, quest'ultimo modifica il primo nel senso che cancella quelle tuple di valori non permesse dalle regole di rilassamento che sono state applicate.

Infine, un terzo tipo di programma logico, detto *solution program*, realizza l'intero algoritmo di soluzione della rete dando il controllo (sul *network program* che rappresenta la rete da risolvere) prima al *relaxation program* e poi all'interprete standard (perchè venga utilizzato il backtracking).

Inoltre, l'uso degli algoritmi di rilassamento in programmazione logica può essere visto come un metodo generale per trasformare un qualsiasi programma logico con la forma di un *network program* in un nuovo programma con le stesse caratteristiche ma più facile da eseguire.

In particolare, è possibile caratterizzare una classe di programmi logici (*grammar logic programs*) per cui esiste sempre la possibiltà di usare un algoritmo di rilassamento perfetto: questi programmi possono essere elaborati prima da un'esecuzione simbolica top-down, e poi da un'esecuzione bottom-up (questo è l'algoritmo di rilassamento perfetto). Tale elaborazione è inoltre molto efficiente (lineare nel numero delle clausole della fase top-down).

Abbiamo scelto la programmazione logica per esprimere le reti di vincoli

e gli algoritmi di rilassamento per le seguenti ragioni:
- la sua forma relazionale, che rende naturale la definizione dei vincoli;
- le clausole definite, che sono adatte per esprimere sia le reti che le regole di rilassamento;
- la possibiltà di esprimere una strategia di controllo nello stesso linguaggio, semplicemente scrivendo un metaprogramma che agisce su un programma a livello oggetto; questo dà anche una grande flessibilità nel poter usare differenti strategie di controllo (cioè differenti algoritmi di rilassamento);
- il funzionamento dell'interprete standard, che usa il backtracking e quindi può essere direttamente usato come algoritmo di soluzione per le reti, semplicemente scrivendo un *network program* che definisce la rete da risolvere.

Un altro tentativo di usare la programmazione logica per risolvere efficientemente le reti di vincoli è dovuto a Van Hentenryck, che, in [15], integra alcune tecniche di consistenza (forward checking, look ahead, partial look ahead) nella programmazione logica estendendo la risoluzione SLD per utilizzarle, e introduce il concetto di dominio estendendo l'unificazione per trattarlo. Con questi nuovi strumenti, l'utente può scrivere un semplice programma che descrive la rete da risolvere, e il nuovo interprete userà automaticamente le tecniche di consistenza, durante il backtracking, per risolverla. Si noti che l'approccio di Van Hentenryck è basato sull'estensione del linguaggio per permettere l'uso delle tecniche di consistenza desiderate, mentre nel nostro approccio le tecniche di consistenza sono definite mediante metaprogrammi. Tale differenza è dovuta principalmente al fatto che Van Hentenryck considera quelle tecniche da applicare durante il backtracking, mentre noi consideriamo quelle da applicare prima.

La sezione 2 definisce le reti di vincoli in termini di ipergrafi di connessione etichettati, la sezione 3 traduce le reti in programmi logici, la sezione 4 introduce gli algoritmi di rilassamento e dà una valutazione della loro complessità, la sezione 5 mostra come scrivere gli algoritmi di rilassamento in programmazione logica. Infine, la sezione 6 dà un'interpretazione degli algoritmi di rilassamento in programmazione logica come un metodo generale di trasformazione di *network programs*, o di programmi che possono essere facilmente ridotti a *network programs*, in nuovi programmi logici che possono essere eseguiti molto più efficientemente, mentre la sezione 7 conclude il lavoro.

Tutte le prove dei teoremi presenti nel lavoro sono state omesse per ragioni di spazio (la versione completa è in [13]). Tutte le parole inglesi sono scritte in corsivo, tranne quelle che sono ormai entrate nell'uso comune del linguaggio dell'informatica.

2. Reti di vincoli

In questa sezione introduciamo gli ipergrafi di connessione e le reti di vincoli. Per una definizione classica degli ipergrafi, si guardi [1], mentre per una descrizione più dettagliata delle reti di vincoli e delle loro proprietà si guardino [10,12].

Definizione 2.1. *(ipergrafi)*
Un **ipergrafo** $G = \langle N,A,c\rangle$ consiste di
- un insieme di **nodi** N;
- un insieme di **iperarchi** A; $A = \cup_k A_k$ e $a \in A_k$ implica che a è un iperarco che connette k nodi (cioè ha rango k);
- una **funzione di connessione**
  $c : \cup_k (A_k \rightarrow N^k)$
  dove $c(a) = \langle x_1,\ldots x_k\rangle$ è la tupla di nodi connessi da a. ♦

Per esempio, consideriamo il semplice ipergrafo $G = \langle N,A,c\rangle$, dove:
- $N = \{x,y,z,w\}$;
- $A = \{a,b,c,d,e\}$;
- $c(a) = \langle x,y,z,w\rangle$, $c(b) = \langle x,y\rangle$, $c(c) = \langle y,z\rangle$, $c(d) = \langle z,w\rangle$, $c(e) = \langle w,x\rangle$.

Questo ipergrafo può essere visto in Fig.1.

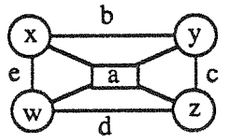

Fig.1. Un semplice ipergrafo.

Definizione 2.2 *(ipergrafi di connessione)*
Un **ipergrafo di connessione** $a \leftarrow G$ è un ipergrafo $G = \langle N,A,c\rangle$ più uno dei suoi iperarchi a, detto **iperarco di connessione**. Se $a \in A_k$, allora G è detto un ipergrafo di k-connessione. Se $c(a) = \langle x_1,\ldots x_k\rangle$, $A\text{-}a = \{a_1,\ldots,a_m\}$ e $c(a_i) = \langle x^i_1,\ldots, x^i_{k(i)}\rangle$, per $i=1,\ldots,m$, allora rappresenteremo l'ipergrafo di k-connessione scrivendo $a(x_1,\ldots x_k) \leftarrow a_1(x^1_1,\ldots,x^1_{k(1)}),\ldots,a_m(x^m_1,\ldots, x^m_{k(m)})$. ♦

Per esempio, consideriamo l'ipergrafo G di Fig.1 dove a è scelto come iperarco di connessione. In questo caso, otteniamo l'ipergrafo di connessione $a \leftarrow G$, che scriveremo $a(x,y,z,w) \leftarrow b(x,y), c(y,z), d(z,w), e(w,x)$.

Definizione 2.3. *(sottoipergrafi)*
Dati due ipergrafi di connessione $a \leftarrow G$ e $b \leftarrow F$, $b \leftarrow F$ è un **sottoipergrafo** di $a \leftarrow G$ se:
- $A_F \subseteq A_G$ (ricordiamo che $b \in A_F$);
- $N_F \subseteq N_G$;
- c_F coincide con c_G su A_F. ♦

Definizione 2.4. *(reti di vincoli)*
Una **rete di vincoli** $C = a \leftarrow G \mid l$ è un ipergrafo di connessione $a \leftarrow G$, dove i nodi sono detti **variabili** e gli iperarchi sono detti **vincoli**, più una **funzione di etichettamento** $l: \cup_k (A_k \rightarrow \wp(U^k))$, dove U è un insieme finito di valori per le variabili di C. ♦

Definizione 2.5. *(soluzione di una rete di vincoli)*
Data una rete di vincoli $C = a \leftarrow G \mid l$ dove $G = \langle N,A,c\rangle$ e $n = \#N$, consideriamo un ordinamento delle variabili di N, sia $\langle x_1,\ldots,x_n\rangle$. Inoltre, data una n-upla $\langle v_1,\ldots,v_n\rangle = \mathbf{v}$ di valori di U, sia $\mathbf{v}\mid\langle x_{i1},\ldots x_{im}\rangle = \langle v_{i1},\ldots,v_{im}\rangle$. Definiamo allora la soluzione della rete C come $Sol(C) =$

$\{<v_1,...,v_n> |_{c(a)}$ tale che, per ogni $b \in A$, $<v_1,...,v_n> |_{c(b)} \in l(b)\}$. ♦

Informalmente, la soluzione di una rete di vincoli è l'insieme di tutti gli assegnamenti delle variabili connesse dall'iperarco di connessione tali che ognuno di questi assegnamenti può essere esteso ad un assegnamento di tutte le variabili di N che soddisfa tutti i vincoli.

Un algoritmo ovvio, che utilizza la ricerca esaustiva per ottenere la soluzione Sol(C) di una rete di vincoli $C = a \leftarrow G \mid l$ è il seguente.

=== Algoritmo ES ===
- Sol(C):=ø.
- Per ogni possibile assegnamento v di tutte le variabili in N:
  - Se, per ogni vincolo b in A, si ha che $v |_{c(b)} \in l(b)$, allora aggiungi $v |_{c(b)}$ a Sol(C).

La complessità in tempo nel caso pessimo di tale algoritmo è $O(\#U^{\#N} \#A \#N)$. Infatti, il numero dei possibili assegnamenti delle variabili su U è $\#U^{\#N}$; per ogni assegnamento, #A vincoli sono verificati, e ogni verifica coinvolge, nel caso pessimo, #N variabili.

Definizione 2.6. *(reti di vincoli risolte)*
Una rete di vincoli $C = a \leftarrow G \mid l$ è **risolta** sse l(a) = Sol(C). ♦

Usando l'algoritmo ES, un modo banale per risolvere una rete di vincoli $C = a \leftarrow G \mid l$ consiste nell'ottenere Sol(C) con questo algoritmo e poi porre l(a) = Sol(C). Infatti, la rete ottenuta $C_{sol} = a \leftarrow G \mid l\ [Sol(C) / a]$ è una rete risolta.

3. Reti di vincoli e programmazione logica

In questa sezione mostriamo come rappresentare le reti di vincoli in programmazione logica. Per un'introduzione alla programmazione logica, si guardi [6].

Per esprimere le reti di vincoli introduciamo certi programmi logici detti *network programs*.

Definizione 3.1. *(network programs)*
Consideriamo un universo di Herbrand universe U. Un ***network program*** è un programma logico che ha un goal atomico, una regola detta **body** e un insieme di fatti. ♦

Data una rete di vincoli, la seguente definizione ci permette di scrivere il corrispondente *network program*. Inoltre, il teorema 3.3 dice che la soluzione della rete coincide con la semantica standard del corrispondente programma.

Definizione 3.2. *(dalle reti ai network programs)*
Data una rete di vincoli $C = a \leftarrow G \mid l$ dove $G = <N,A,c>$, il corrispondente *network program* P è costruito come segue:
- le variabili coincidono con le variabili in N;
- i simboli di predicato coincidono con i vincoli in A;
- il goal è
  $\leftarrow a(u_1,...,u_k)$, dove $<u_1,...,u_k> = c(a)$;

- il body è la regola $a \leftarrow G$, i.e.
  $a(u_1,...,u_k) \leftarrow \wedge \{h(x_1,...,x_i) \mid h \in A\text{-}a, <x_1,...,x_i> = c(h)\}$,
  dove $<u_1,...,u_k> = c(a)$;
- l'insieme dei fatti è
  $\{h(c_1,...,c_i) \mid h \in A\text{-}a, <c_1,...,c_i> \in l(h)\}$. ♦

Teorema 3.3. *(i network programs sono equivalenti alle reti)*
Data una rete C, la sua soluzione coincide con la semantica del corrispondente *network program* P. ♦

Consideriamo la semplice rete $C = a \leftarrow G \mid l$ dove $a \leftarrow G$ è come in Fig.1. Il corrispondente *network program* è:
- il goal è
  $\leftarrow a(x,y,z,w)$;
- il body è
  $a(x,y,z,w) \leftarrow b(x,y), c(y,z), d(z,w), e(w,x)$;
- i fatti sono definiti dalla funzione l; se, per esempio, U = {T,F} e
  $l(b) = \{<T,T>,<F,F>\}$, $l(c) = \{<T,F>,<F,T>\}$, $l(d) = \{<T,T>,<T,F>\}$, $l(e) = \{<F,T>,<F,F>\}$,
  abbiamo
  $b(T,T) \leftarrow$
  $b(F,F) \leftarrow$
  $c(T,F) \leftarrow$
  $c(F,T) \leftarrow$
  $d(T,T) \leftarrow$
  $d(T,F) \leftarrow$
  $e(F,T) \leftarrow$
  $e(F,F) \leftarrow$.

4. Algoritmi di rilassamento

In questa sezione definiamo gli algoritmi di rilassamento sulle reti di vincoli in termini di cambiamenti locali alla funzione di etichettamento.

Data una funzione totale $l : A \rightarrow B$, la notazione $l[a / b]$, dove $a \in B$ e $b \in A$, indicherà la nuova funzione l' tale che $l'(b) = a$, e $l'(c) = l(c)$ per ogni $c \in A$, $c \neq b$. Usando questa notazione, possiamo ora definire le regole di rilassamento, che sono il meccanismo di base degli algoritmi di rilassamento.

Definizione 4.1. *(regole di rilassamento)*
Data una rete di vincoli $C = a \leftarrow G \mid l$, una regola di rilassamento è un qualsiasi sottoipergrafo $b \leftarrow F$ di $a \leftarrow G$. L'applicazione di una regola di rilassamento $b \leftarrow F$ a $C = a \leftarrow G \mid l$ produce una nuova rete di vincoli $C' = a \leftarrow G \mid l\ [Sol(b \leftarrow F \mid l) / b]$. ♦

Informalmente, l'applicazione di una regola di rilassamento $b \leftarrow F$ a $C = a \leftarrow G \mid l$ implica la risoluzione di $b \leftarrow F \mid l$ e il cambiamento di l ottenuto ponendo l'etichetta dell'iperarco di connessione di F uguale alla soluzione di $b \leftarrow F \mid l$.

Teorema 4.2. *(una regola di rilassamento produce una rete equivalente)*
Data una rete di vincoli $C = a \leftarrow G \mid l$ e una regola di rilassamento $b \leftarrow F$, abbiamo:

Sol (a ← G | l) = Sol (a ← G | l [Sol (b ← F | l) / b]). ♦

Definizione 4.3. *(rete stabile)*
Dato un insieme R di regole di rilassamento, una **rete stabile** rispetto a R è una rete C = a ← G | l tale che, per ogni r in R, l'applicazione di r a C produce C. ♦

Per esempio, la rete C = a ← G | l, dove l(h)=∅ per ogni h in A, è una rete stabile rispetto ad un qualunque insieme di regole di rilassamento. Lo stesso è per C = a ← G | l dove, per ogni h in A, se $h \in A_k$, allora $l(h) = U^k$.

Definizione 4.4. *(chiusura di una rete, algoritmo di rilassamento)*
Data una rete di vincoli C = a ← G | l, un insieme R di regole di rilassamento e una stringa $S \in R^\infty$ dove ogni regola di R appare un numero infinito di volte (S è detta una **strategia** per R), la **chiusura** di C rispetto a S è una rete C' = a ← G | l', stabile rispetto a R, che è ottenuta applicando le regole in S fino a raggiungere la stabilità. Scriveremo C' = chiusura(C,S) e diremo **algoritmo di rilassamento** l'algoritmo che calcola C'. ♦

Teorema 4.5. *(la chiusura non dipende da S)*
Data una rete di vincoli C = a ← G | l e un insieme R di regole di rilassamento, consideriamo due strategie S' e S" per R. Abbiamo che chiusura(C,S') = chiusura(C,S"). Quindi, scriveremo **chiusura(C,R)**. ♦

Il teorema 4.5 può essere provato usando alcuni risultati sulla teoria dei lattici ([14]) e sull'iterazione caotica ([2]).
Il teorema 4.6 segue invece da ripetute applicazioni del teorema 4.2.

Teorema 4.6. *(una rete è equivalente alla sua chiusura)*
Data una rete di vincoli C = a ← G | l e un insieme R di regole di rilassamento, abbiamo: Sol(C) = Sol(chiusura(C)). ♦

Definizione 4.7. *(insiemi adeguati di regole)*
Data una rete di vincoli C = a ← G | l e un insieme R di regole di rilassamento, R is **adeguato** per C sse la rete C' = chiusura(C,R) è risolta. ♦

Dal teorema 4.2 e dalla definizione 4.7 segue che, data una rete di vincoli C e un insieme R di regole di rilassamento adeguato per C, l'etichetta dell'iperarco di connessione in C' = chiusura(C,R) è la soluzione di C.

Come esempio di algoritmo di rilassamento, consideriamo l'algoritmo di *arc-consistency* proposto da Mackworth in [6] per reti di vincoli binarie; nella nostra notazione, la rete C = a ← G | l è tale che G contiene tutti i vincoli unari a_i, con $c(a_i) = \langle x_i \rangle$, più alcuni vincoli binari a_{ij}, con $c(a_{ij}) = \langle x_i,x_j \rangle$. R = { $a_i(x_i) \leftarrow a_{ij}(x_i,x_j), a_j(x_j)$, per i,j=1,...,#N} e in generale non è adeguato per C. Una generica regola di rilassamento dell'algoritmo di *arc-consistency* può essere vista in Fig.2 (l'iperarco di connessione è rappresentato da una linea più spessa).

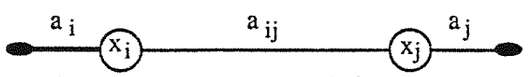

Fig.2. Una regola di rilassamento per l'algoritmo di *arc-consistency*.

$$SS(f) = f'$$

tale che se l'arietà di f è n, f' è una funzione (parziale o totale)

$$f': D_{SS}^n \dashrightarrow D_{SS};$$

F_{SS} è l'isieme dei simboli di funzione denotati;

3) per ogni $p \in P_{SS}$ con $P(L) \supseteq P_{SS}$ (P_{SS} è un sottinsieme dei simboli di predicato) è definito un attachment:

$$SS(p) = p'$$

tale che se l'arietà di p è n, p' è un predicato (parziale o totale)

$$p': D_{SS}^n \dashrightarrow \{V, F\};$$

P_{SS} è l'insieme dei simboli di predicato denotati.

Oss 2.1: Si osservi che un attachment SS su D_{SS} definisce una struttura di simulazione

$$SS' = \langle D_{SS}, F_{SS}', P_{SS}', C_{SS}' \rangle$$

dove

$F_{SS}' = \{f' \mid f' = SS(f)$ per qualche $f \in F_{SS}\}$

$P_{SS}' = \{p' \mid p' = SS(p)$ per qualche $p \in P_{SS}\}$

$C_{SS}' = \{c' \mid c' = SS(c)$ per qualche $c \in C_{SS}\}$

D'ora in poi non verrà più distinto SS come attachment da SS' come struttura determinata da tale attachment.

Al fine di familiarizzare il lettore con il concetto di struttura di simulazione, proponiamo un esempio del suo uso in FOL.

Esempio 2.1: Pensiamo a un linguaggio L caratterizzato da un insieme di costanti {0, 1, 2, ...}, un insieme di funzioni {p} ed un insieme di relazioni {=, <}. Definiamo quindi la struttura di simulazione :

SS(0) = $\underline{0}$, SS(1) = $\underline{1}$, SS(2) = $\underline{2}$

SS(p) (x) = pred(x) , SS (=)(x, y) = eq(x,y),

SS(<) (x,y) = T se x = pred (y), F altrimenti

Parafrasando, i numerali 0, 1, 2 sono associati ai numeri $\underline{0}$, $\underline{1}$, $\underline{2}$, il simbolo di funzione p è associato alla usuale funzione predecessore pred, e analogamente per i simboli di predicato = e <.

Notiamo che p, = sono totali, e che < è un predicato parziale. Stabiliamo quindi che la teoria sia caratterizzata dal solo assioma $\forall x \forall y \forall z. (x<y \wedge y<z \rightarrow x<z)$. Come conseguenza abbiamo che la formula atomica *0<2* non può venir dedotta ne' dalla conoscenza sintattica ne' da quella semantica espressa dalla struttura di simulazione. Abbiamo però la possibilità di ottenere il risultato sfruttando ambedue i tipi di conoscenza, nel seguente modo:

a) 0<1 per l'attachment di <, 0, 1.

Consideriamo ora un tipo particolare di reti C = a ← G | l dove a ← G è definito come segue:
$a(x,y) \leftarrow b_0(x,u_1), b_1(u_1,u_2), b_2(u_2,u_3), b_3(u_3,y), c_0(x,v_1), c_1(v_1,v_2), c_2(v_2,v_3), c_3(v_3,y), d_1(u_1,v_1), d_2(u_2,v_2), d(u_3,v_3), e_1(x,u_1,v_1), e_2(x,u_2,v_2), e_3(x,u_3,v_3)$.
La funzione l è:
$l(e_1) = l(e_2) = l(e_3) = U^3$, $l(a) = U^2$, $l(b_i)$, $l(c_i)$ e $l(d_i)$, i=0,...,3 sono definite in G in qualche modo.
La rete C appena definita (senza gli iperarchi e_i, i=1,...,3 per rendere leggibile la figura) può essere vista in Fig.6.

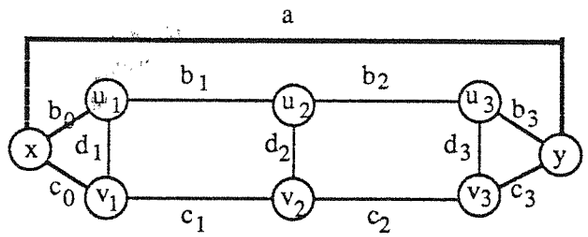

Fig.6. Una rete di vincoli.

Per questa rete, consideriamo le seguenti regole di rilassamento:
$e_1(x,u_1,v_1) \leftarrow b_0(x,u_1), c_0(x,v_1), d_1(u_1,v_1)$.
$e_i(x,u_i,v_i) \leftarrow e_{i-1}(x,u_{i-1},v_{i-1}), b_i(u_{i-1},u_i), c_i(v_{i-1},v_i), d_i(u_i,v_i)$. per i=2,3
$a(x,y) \leftarrow e_3(x,u_3,v_3), b_3(u_3,y), c_3(v_3,y)$.

Si può provare che questo insieme di regole è adeguato per la rete data e che inoltre la rete stabile può essere ottenuta applicando ogni regola una sola volta, nell'ordine in cui le regole sono scritte. Tale insieme di regole è detto perfetto.

Questa stessa proprietà vale anche per tutte le reti ottenute da quella di Fig.6 estendendo 3 ad un generico n.

Definizione 4.8. *(insiemi perfetti di regole)*
Data una rete di vincoli C e un insieme R di regole di rilassamento (#R = n), R è **perfetto** per C sse R è adeguato (C' = chiusura(C,R) è risolta) ed esiste una stringa $S \in R^n$ in cui ogni regola di R appare esattamente una volta, tale che C' può essere ottenuta applicando le regole di S. S è detta una **strategia perfetta** per R. L'algoritmo che applica le regole di S ottenendo C' è detto **algoritmo di rilassamento perfetto.** ♦

In pratica, S non è altro che una versione ordinata di R.

E' possibile caratterizzare quelle classi di reti a cui è possibile applicare un algoritmo di rilassamento perfetto in modo efficiente: sono le classi di reti generabili da una grammatica libera da contesto su ipergrafi (le produzioni di tali grammatiche rimpiazzano un iperarco con un intero ipergrafo). Infatti, l'insieme perfetto di regole di rilassamento per ognuna di tali reti consiste delle produzioni usate per generarla, e la strategia perfetta presenta le regole nell'ordine inverso in cui sono state usate nella fase di generazione. Per una descrizione più dettagliata e formale delle reti di vincoli generate da grammatiche su ipergrafi libere da contesto, si guardi [11].

Cerchiamo ora di studiare la complessità in tempo degli algoritmi di rilassamento.

In generale, la complessità di un qualsiasi algoritmo di rilassamento che soddisfa la definizione 4.4 dipende dal numero di regole che vengono applicate, cioè da quanta parte della strategia S deve essere effettivamente usata per raggiungere la stabilità. Supponendo di applicare r regole, dove la regola i contiene n_i variabili, allora la complessità sarà $O(S_{i=1,\ldots,r} \#U^{n_i})$. Ora, n_i, una volta dato R, è limitato rispetto al numero n di variabili della rete (quindi anche $\#U^{n_i}$ è una quantità limitata), ma r può essere anche molto grande.

Per esempio, l'algoritmo di arc-consistency ha una complessità in tempo $O(\#A\ \#U^2)$ e l'algoritmo di path-consistency $O(\#N^3\ \#U^3)$ (si guardino [8,9]).

Per gli algoritmi di rilassamento perfetti, invece, r si riduce a #R, perciò la complessità in tempo è $O(S_{i=1,\ldots,\#R} \#U^{n_i})$. Ora mostreremo che questa quantità è sempre lineare nel numero dei vincoli della rete da risolvere.

Consideriamo infatti le classi di reti generabili da una grammatica libera da contesto su ipergrafi, in cui ogni regola di rilassamento deriva dall'applicazione di una produzione (la regola è b ← F se la produzione ha sostituito l'iperarco b con l'ipergrafo F), e la strategia ordina le regole nell'ordine inverso a quello di applicazione delle produzioni. Questo algoritmo di rilassamento perfetto è molto efficiente nel risolvere le reti di una tale classe. Infatti, le regole sono sempre tante quante le produzioni e ogni regola può essere applicata con una complessità limitata (se la regola è b ← F, la complessità è un esponenziale nel numero di variabili in F). Inoltre, ogni regola è applicata solo una volta, così l'intero algoritmo di rilassamento ha una complessità in tempo lineare nel numero delle regole, cioè nel numero delle produzioni, e quindi lineare nel numero dei vincoli della rete se supponiamo che, nella grammatica considerata, non è possibile riscrivere infinite volte lo stesso iperarco.

Naturalmente, è facile vedere che in realtà, data una qualsiasi singola rete, è sempre possibile trovare una algoritmo di rilassamento perfetto per essa. Tale algoritmo avrà però una complessità nel caso pessimo esponenziale nel size della rete. Se invece si considera un'intera classe di reti generabili da una certa grammatica libera da contesto su ipergrafi, allora l'algoritmo di rilassamento perfetto che risolve tutte le reti della classe ha una complessità nel caso pessimo che rimane lineare nel numero dei vincoli, in quanto non dipende dalla rete da risolvere ma solo dal size delle produzioni, che è dato insieme alla grammatica.

Ad esempio, l'algoritmo di rilassamento perfetto descritto con la Fig.6 usa n+1 regole, ognuna con al più 5 variabili; quindi la sua complessità è $O(n\ \#U^5)$. Ricordiamo che nella classe di reti considerata si ha #A = 3n + 2, quindi la complessità è anche $O(\#A\ \#U^5)$, cioè O(#A) (in questo caso particolare #N = 2n + 2, quindi la complessità è anche O(#N), ma questo non è vero in generale).

5. Algoritmi di rilassamento e metaprogrammazione logica

Nella sezione 3 abbiamo visto che una rete C = a ← G | l può essere rappresentata da un *network program* composto da un goal atomico (← a), una regola (a ←G) e un insieme di fatti che definiscono la funzione l. Inoltre, la semantica di questo *network program* è la soluzione esatta della rete C = a ←G | l.

In questa sezione vedremo che un algoritmo di rilassamento applicato a C può essere espresso tramite un programma logico che modifica la base di dati

(cioè l'insieme dei fatti che definiscono i vincoli) del *network program* che corrisponde a C.

Infatti, ogni regola di rilassamento b ← F modifica l(b), cioè la definizione del vincolo b in C, rendendolo più vincolante. Quindi, rappresentando b ← F con la corrispondente regola (la cui forma è proprio b ← F), la selezione di questa regola modifica l'insieme dei fatti che definiscono il predicato b, rimuovendone qualcuno (quelli non permessi dalla regola di rilassamento, ciopè dai vincoli di F).

Se le regole di rilassamento usato erano adeguate, la definizione del vincolo a di C coincide con Sol(C). Altrimenti, per ottenere la soluzione della rete, possiamo eseguire (usando l'interprete standard) il *network program* corrispondente a C '= a ← G | l', dove l' è la nuova definizione dei vincoli di C dopo l'applicazione dell'algoritmo di rilassamento.

Supponiamo di avere un algoritmo di rilassamento che usa n regole, dove la regola i ha la forma $b_i \leftarrow F_i$. Il corripondente programma logico, detto ***relaxation program***, può essere così formato:

- una regola $\mathbf{b_i(x_1,...,x_k) \leftarrow F_i}$ per ogni regola di rilassamento $b_i \leftarrow F_i$ tale che $c(b_i)=<x1,...,x_k>$ (se b_i è l'iperarco di connessione della rete, allora gli diamo un altro nome per evitare interferenze tra questa clausola e l'unica regola del *network program* che rappresenta la rete da risolvere);
- una regola **iter ←rule(1),..., rule(n), check** che indica la strategia scelta (per convenienza, abbiamo qui scelto la strategia che ripete all'infinito l'applicazione sequenziale di tutte le regole di R, verificando dopo ogni iterazione se qualcosa è cambiato);
- due clausole
  rule(i) ←not($b_i(x_1,...,x_k)$),retract($b_i(x_1,...,x_k)$),assertz(change(i)),fail
  rule(i) ←
  per ogni regola di rilassamento: esse applicano la regola i ed eliminano dalla base di dati tutti quei fatti non permessi da questa regola;
- due clausole
  check ←change(i), polish, iter
  check ←
  che definiscono la verifica da fare dopo ogni iterazione: la prima regola, che viene usata se qualcosa è cambiato, rimuove i fatti che asseriscono tali cambiamenti, e ripete ancora l'applicazione di tutte le regole; la seconda, che viene usata se non è cambiato niente (si è raggiunta la stabilità), ridà il controllo alla regola che definisce l'iterazione;
- due clausole
  polish ←retract(change(i)), fail
  polish ←
  che rimuovono tutti i fatti che definiscono i cambiamenti effettuati dall'iterazione precedente;
- una regola
  relax_alg ←iter
  che definisce l'algoritmo di rilassamento come l'applicazione iterata di un insieme di regole di rilassamento.
  Il goal di un relaxation program così fatto è **←relax_alg**.

Dato un *network program* (senza il goal) e un *relaxation program*, non otterremo niente di esplicito dall'esecuzione di questo programma, ma come effetto collaterale avremo la cancellazione di alcuni dei fatti del dato *network program*. In questo senso, un *relaxation program* può essere visto come un metodo di trasformazione di un *network program* in un altro *network program* più esplicito, cioè meno ridondante.

Se l'algoritmo di rilassamento da usare non è adeguato, la soluzione della rete può essere ottenuta, come sappiamo, applicando prima questo algoritmo di rilassamento e poi usando il *backtracking* alla rete ottenuta. In programmazione logica questo vuol dire che aggiungiamo al *network program* (senza il goal) e al *relaxation program* il goal **← sol** e le regole
sol ← relax_alg, !, a(x_1,...,x_k), assertz(a(x_1,...,x_k)), fail (una nota: il *cut* è necessario per evitare una nuova applicazione dell'algoritmo di rilassamento dopo aver trovato tutte le soluzioni della rete, e per dare invece il controllo alla regola successiva che lista tutte queste soluzioni);
sol ← listing(a).
Chiamiamo questo insieme di clausole un ***solution program***.

Quindi, la soluzione di una rete C può essere ottenuta, utilizzando la programmazione logica, scrivendo

- il *network program* corrispondente a C (senza il goal);
- il *relaxation program* corrispondente all'algoritmo di rilassamento voluto;
- il *solution program*.

Notiamo che, in questa configurazione (*network program* più *relaxation program* più *solution program*), il *network program* descrive la base di conoscenza e il *relaxation* più il *solution program* cambiano questa base di conoscenza eliminando alcuni fatti ogni volta che viene applicata una regola di rilassamento. In un certo senso, il *solution program* è responsabile del controllo sulla base di conoscenza definita dal *network program*: prima dà questo controllo al *relaxation program* perchè la rete sia esplicitata il più possibile, e poi all'interprete Prolog per elaborare la nuova base di conoscenza con il backtracking.

Consideriamo ora alcuni casi particolari: quando l'insieme di regole di rilassamento è adeguato e quando è perfetto.
Se l'insieme di regole di rilassamento è adeguato, allora possiamo scrivere

- l'insieme dei fatti contenuti nel *network program* (la regola a ← G non è necessaria perchè il backtracking non serve);
- il *relaxation program*;
- il *solution program* semplificato in questo modo
  ←sol
  sol ←relax-alg, listing(a).

Se l'insieme di regole di rilassamento è perfetto, è possibile eliminare alcune clausole anche dal *relaxation program*. Infatti in questo caso le regole sono applicate solo una volta, così la verifica di cambiamenti non è più necessaria. Quindi le clausole che definiscono i predicati "*check*" e "*polish*" possono essere tolte. Inoltre, invece di definire tutti gli iperarchi di connessione (uno per ogni regola) come relazioni totali e poi eliminare tutti quei fatti non permessi dalle regole, possiamo fare il contrario, cioè non definire gli iperarchi di connessione e poi asserire i fatti permessi da ogni regola. Perciò abbiamo (oltre alle clausole che definiscono le regole di rilassamento)

iter ←rule(1),...,rule(n)
rule(i) ←$b_i(x_1,...,x_k)$, assertz($b_i(x_1,...,x_k)$), fail per i=1,...,n
rule(i) ← per i=1,...,n
relax_alg ←iter.

Quindi, per risolvere una rete tramite un algoritmo di rilassamento perfetto

e usando la programmazione logica, possiamo scrivere

- l'insieme dei fatti contenuti nel *network program* (la regola a ← G non è necessaria perchè il backtracking non serve);
- il *relaxation program* senza verifiche di cambiamenti (come appena scritto);
- il *solution program* semplificato in questo modo
  ←**sol**
  sol ←relax–alg, listing(a).

Vediamo ora alcuni esempi.

Consideriamo l'algoritmo di *arc-consistency*, che ha le regole di rilassamento della forma $a_i(x_i) \leftarrow a_{ij}(x_i,x_j),a_j(x_j)$. Il corrispondente *relaxation program* è:

$a_i(x_i) \leftarrow a_{ij}(x_i,x_j),a_j(x_j)$, per ogni i,j in N, a_i,a_j,a_{ij} in A
iter ← rule(1),...,rule(n), check
rule(i) ← not(b_i(x)), retract(b_i(x)), assertz(change(i)), fail per i=1,...,n
rule(i) ← per i=1,...,n
(notiamo che in generale la regola i non definisce l'iperarco a_i, perchè ci sono differenti regole che definiscono a_i, una per ogni a_{ij} in A)
check ← change(i), polish, iter
check ←
polish ← retract(change(i)), fail
polish ←
arc_cons ← iter.

L'intero programma logico necessario per risolvere una rete di vincoli applicando prima l'algoritmo di *arc-consistency* e poi usando il backtracking (le regole di *arc-consistency* non sono adeguate in generale) consiste del *relaxation program* appena scritto, del *network program* corrispondente alla rete da risolvere (senza il goal), e del seguente *solution program*:

sol ← arc_cons, !, a(x_1,...,x_k), assertz(a(x_1,...,x_k)), fail
sol ← listing(a),
← sol.

Come ulteriore esempio, consideriamo l'algoritmo di *path-consistency*, il cui *relaxation program* corrispondente è

$a_{ij}(x_i,x_j) \leftarrow a_{ik}(x_i,x_k),a_k(x_k), a_{kj}(x_k,x_j)$, per ogni i,j,k in N
iter ← rule(1),...,rule(n), check
rule(i) ← not(b_i(x,y)), retract(b_i(x,y)), assertz(change(i)), fail per i=1,...,n
rule(i) ← per i=1,...,n
check ← change(i), polish, iter
check ←
polish ← retract(change(i)), fail
polish ←
path_cons ← iter.

Anche in questo caso, l'intero programma logico necessario per risolvere una rete di vincoli applicando prima l'algoritmo di *path-consistency* e poi usando il backtracking (le regole di *path-consistency* non sono adeguate in generale) consiste del *relaxation program* appena scritto, del *network program* corrispondente alla rete da risolvere (senza il goal), e del seguente *solution program*:

sol ← path_cons, !, a(x_1,...,x_k), assertz(a(x_1,...,x_k)), fail
sol ← listing(a),
← sol.

Infine, consideriamo la classe delle reti come quella in Fig.6 e le regole di rilassamento descritte per tale classe:

$e_1(x,u_1,v_1) \leftarrow b_0(x,u_1), c_0(x,v_1), d_1(u_1,v_1)$
$e_i(x,u_i,v_i) \leftarrow e_{i-1}(x,u_{i-1},v_{i-1}), b_i(u_{i-1},u_i), c_i(v_{i-1},v_i), d_i(u_i,v_i)$ per i=2,...,n
$a(x,y) \leftarrow e_n(x,u_n,v_n), b_n(u_n,y), c_n(v_n,y)$.

Sappiamo già che questo è un insieme perfetto di regole di rilassamento, quindi il programma logico per risolvere una di queste reti consiste dei fatti che definiscono i vincoli delle rete, del seguente *relaxation program*

$a(x,y) \leftarrow e_n(x,u_n,v_n), b_n(u_n,y), c_n(v_n,y)$
$e_i(x,u_i,v_i) \leftarrow e_{i-1}(x,u_{i-1},v_{i-1}), b_i(u_{i-1},u_i), c_i(v_{i-1},v_i), d_i(u_i,v_i)$ per i=n,...,2
$e_1(x,u_1,v_1) \leftarrow b_0(x,u_1), c_0(x,v_1), d_1(u_1,v_1)$
iter ← rule(1),...,rule(n+1)
rule(i) ← e_i(x,u,v), assertz(e_i(x,u,v)), fail per i=1,...,n
rule(i) ← per i=1,...,n
rule(n+1) ← a(x,y), assertz(a(x,y)), fail per i=1,...,n
rule(n+1) ←
relax_alg ← iter,

e del seguente *solution program*

sol ← relax_alg, listing(a)
← sol.

6. Algoritmi di rilassamento in programmazione logica: una metodologia di trasformazione di programmi logici

La sezione 3 mostra come si può risolvere una rete di vincoli nel contesto della programmazine logica usando il backtracking: la rete viene rappresentata da un *network program* (una regola più un insieme di fatti, il cui universo di Herbrand è un insieme finito di costanti), l'esecuzione del quale produce la soluzione della rete.

Nella sezione 4 si vede che un algoritmo di rilassamento prende una rete e ne produce un'altra, equivalente a quella data (cioè che ha la stessa soluzione), ma più esplicita, cioè più vicina ad essere risolta.

Infine, la sezione 5 propone di rappresentare ogni algoritmo di rilassamento nel contesto della programmazione logica con un corrispondente *relaxation program*, che è un programma logico che, eseguito assieme ad un *network program*, modifica la parte estensionale di quest'ultimo cancellando alcuni fatti.

Quindi, un *relaxation program* può essere visto come un metaprogramma il cui input è un *network program* e il cui output è un altro *network program* che è una trasformazione dell'input. Se ora paragoniamo le esecuzioni dei due *network programs* che sono rispettivamente l'input e l'output di un *relaxation program*, si vedrà che il primo richiede, in generale, una maggiore elaborazione, nel senso che l'algoritmo di backtracking fallirà molto di più prima di ottenere una soluzione della rete corrispondente. Quindi, i *relaxation programs* possono essere considerati un metodo per migliorare i *network programs* rispetto alla complessità della loro esecuzione. Questo miglioramento può essere naturalmente esteso ad un qualsiasi programma logico la cui struttura è quella di un *network program* (cioè non necessariamente dobbiamo considerare programmi costruiti a partire da reti di vincoli), o che ha uno o più sottoprogrammi con tale forma.La situazione è ancora più promettente se si considerano *relaxation programs* che rappresentano algoritmi di rilassamento perfetti. In questo

caso, infatti, il network program ottenuto rappresenta una rete risolta, e quindi non ha più bisogno di nessuna elaborazione.

Gli algoritmi di rilassamento perfetti sono anche efficienti se vengono applicati a quelle classi di reti incluse nel linguaggio di qualche grammatica libera da contesto su ipergrafi. Per ogni rete della classe, si usano come regole di rilassamento le produzioni usate per generarla, e tali regole si applicano con un ordine che è l'inverso di quello usato nella fase di generazione.

Questa situazione può essere trasporata nella programmazione logica, se consideriamo *network programs* corrispondenti a reti appartenenti alle classi ora descritte, e *relaxation programs* corrispondenti ai relativi algoritmi di rilassamento perfetti.

Per fare questo definiamo ora la classe dei programmi logici che corrispondono alle grammatiche libere da contesto su ipergrafi, e chiamiamo tali programmi *grammar logic programs*. Ognuno di questi programmi rappresenta un insieme di *network programs* (quelli corrispondenti alle reti generabili dalla grammatica).

Definizione 6.1. *(grammar logic programs)*
Un ***grammar logic program*** è un programma logico con queste caratteristiche:

- l'universo di Herbrand è un insieme finito di costanti;
- le regole contengono solo variabili;
- i fatti sono ground;
  quello dei predicati definiti dai fatti, detti predicati estensionali, sono disgiunti;
- il goal è atomico e relativo ad uno dei predicati intensionali. ♦

La relazione tra una grammatica libera da contesto su ipergrafi e un *grammar logic program* è semplice e può essere informalmente descritta così: i predicati intensionali corrispondono agli iperarchi non terminali, mentre quelli estensionali corrispondono ai terminali, ogni regola rappresenta una produzione (la parte sinistra è relativa all'iperarco riscritto e quella destra all'ipergrafo generato), i fatti definiscono i predicati terminali (cioè i vincoli delle reti generabili), il goal rappresenta il simbolo iniziale della grammatica.

Se l'esecuzione di un *grammar logic program* viene effettuata usando la normale risoluzione SLD con una regola di computazione che seleziona sempre, nel goal corrente, un atomo con un predicato intensionale, allora una qualsiasi derivazione SLD terminerà sempre, con fallimento, con una collezione di sottogoal $(A_1,...,A_n)$ i cui predicati sono tutti estensionali. Quindi, il risultato di ogni computazione è la clausola $(G \leftarrow A_1,...,A_n)\sigma$, dove G è il goal dato, σ è la sostituzione ottenuta durante la derivazione SLD, $A_i = P_i(x_{i1},...,x_{im(i)})$ e P_i è un predicato estensionale per i=1,...,n. E' facile vedere che questa clausola, assieme alla parte estensionale (i fatti) del *grammar logic program* di partenza (ristretta a quei predicati estensionali che appaiono nella clausola stessa), costituisce un nuovo programma che ha la stessa forma di un *network program*. Questo *network program* definisce una delle reti generabili dalla grammatica corrispondente al *grammar logic program* che è stato eseguito.

Come sappiamo, per l'esecuzione di questo *network program*, possiamo usare il normale backtracking oppure un algoritmo di rilassamento. Ma, se abbiamo conservato la sequenza di clausole usate durante la derivazione SLD, possiamo usare l'algoritmo di rilassamento perfetto le cui regole corrispondono a tali clausole e in cui l'ordine di applicazione è l'inverso di

quello usato nella fase di derivazione. Alla fine di questo algoritmo di rilassamento perfetto, che può essere facilmente scritto sotto forma di *relaxation program*, avremo tutte le istanziazioni ground del goal, cioè la soluzione della rete corrispondente al *network program* eseguito.

Si noti che questo algoritmo di rilassamento realizza un tecnica di esecuzione bottom-up, che è molto semplice e naturale da implementare nel caso dei *grammar logic programs* a causa della loro particolare struttura.

Per generalizzare dal contesto delle reti di vincoli, se abbiamo un programma logico con la forma di un *grammar logic program*, possiamo prima usare la regola di computazione sopra descritta, ottenere così un *network program*, e poi eseguirlo con un algoritmo di rilassamento perfetto (relativo alle clausole usate nella prima fase dell'esecuzione). In questo modo avremo un insieme di conseguenze logiche del programma logico considerato con una complessità in tempo che è lineare nel numero di clausole usate per generare il *network program*.

E' interessante notare che la stessa idea (di un'esecuzione bottom-up) può essere anche usata per una esecuzione efficiente di un *network program*. Infatti, consideriamo un *network program* che rappresenta la rete $C = a^{\cdot} G \mid l$, dove $G=\langle N,A,c\rangle$ e $A=\{h_1,...,h_m\}$. E' facile vedere che un'esecuzione *bottom-up* di questo programma coincide con la computazione dell'espressione dell'algebra relazionale $proj_{c(a)}$ (join($l(h_1)$, $l(h_2)$, ... $l(h_m)$)), che calcola il *join* naturale delle relazioni che descrivono i vincoli della rete e poi proietta la relazione così ottenuta sulle variabili connesse dall'iperarco di connessione. La complessità di tale esecuzione è $O(\#l(h_1)\ \#l(h_2)\ ...\#l(h_m))$, dove $\#l(h_i)$ è il numero di tuple in $l(h_i)$.

7. Conclusioni

Abbiamo mostrato come usare la programmazione logica per risolvere le reti di vincoli, sia mediante il backtracking che utilizzando degli algoritmi di rilassamento.

Data una rete di vincoli, possiamo facilmente scrivere un corrispondente semplice programma logico (*network program*) la cui semantica coincide con la soluzione della rete: in questo caso la rete viene risolta usando solo il backtracking.

Se invece vogliamo usare anche un algoritmo di rilassamento, possiamo aggiungere al *network program* un altro programma logico (*relaxation program*) il cui compito è quello di modificare la parte estensionale del *network program* (cioè la definizione dei vincoli della rete) per raggiungere un certo grado di consistenza locale nella rete. Questo *relaxation program* può essere considerato un metaprogramma logico che realizza una pre-elaborazione del *network program* prima di consegnarlo all'interprete standard: in questo caso la rete viene risolta prima applicando delle tecniche di rilassamento e poi il backtracking.

Questo risultati possono essere generalizzati, poichè l'uso di algoritmi di rilassamento in programmazione logica può essere visto come un metodo generale per trasformare un qualsiasi programma logico con la forma di un *network program* in un nuovo programma che ha le stesse caratteristiche ma che è più semplice da eseguire.

References

1. Berge C., "Graphs and Hypergraphs", North Holland, Amsterdam, The Netherlands, 1973.

2. Cousot P., Cousot R., "Constructive version of Tarski's fixed point theorems", *Pacific J. Math. 82*, 1, 1979, pp.1-12.

3. Dechter R., Pearl J., "Network-based heuristics for constraint-satisfaction problems",*Technical Report R-90, UCLA,* Cognitive Systems Lab., Los Angeles, Cal., May 1987.

4. Freuder E.C., "Synthesizing constraint expressions", *Communication ACM 21*, 11, Nov. 1978, pp.958-966.

5. Freuder E.C., "A sufficient condition for backtrack-free search", *Journal ACM 29*, 1, Jan. 1982, pp.24-32.

6. Lloyd J.W., *"Foundations of logic programming"*, Springer-Verlag, 1984.

7. Mackworth A.K., "Consistency in networks of relations", *Artificial Intelligence 8*, 1, 1977, pp.99-118.

8. Mackworth A.K., Freuder E.C., "The complexity of some polynomial network consistency algorithms for constraint satisfaction problems", *Artificial Intelligence 25*, 1985, pp.65-74.

9. Mohr R., Henderson T.C., "Arc and Path Consistency Revisited", *Artificial Intelligence 28*, 1986, pp.225-233.

10. Montanari U., "Networks of constraints: fundamental properties and application to picture processing", *Information Science 7*, 1974, pp.95-132.

11. Montanari U., Rossi F., "An efficient algorithm for the solution of hierarchical networks of constraints", apparirà sui *Proc. Int. Workshop on Graph Grammars and Their Application to Computer Science*, Warrenton, Dec. 1986, Ehrig ed., LNCS, Springer-Verlag.

12. Montanari U., Rossi F., "Fundamental properties of networks of constraints: a new formulation", apparirà su *"Search and Artificial Intelligence"*, Kanal ed., Springer-Verlag.

13. Rossi F., Montanari U., "Relaxation in networks of constraints as higher order logic programming", Nota Interna, Dipartimento di Informatica, Università di Pisa, Pisa, Italia, 1988.

14. Tarski A., "A lattice-theoretical fixpoint theorem and its applications", *Pacific J. Math. 5*, 1955, pp.285-309.

15. Van Hentenryck P., "A theoretical framework for consistency techniques in logic programming", *Proc. Tenth IJCAI*, Milan, August 1987, pp.2-8.

Temporal Reasoning in a Hybrid System

M. Poesio,

University of Hamburg, Computer Science Department,

Jungiusstr. 6, 2000 Hamburg 36,

Federal Republic of Germany

Abstract

Temporal aspects of knowledge are receiving increasing attention for their importance both in organizing knowledge bases and in natural language understanding. Most of the current proposals follow a purely logical approach without taking advantage of the possibilities offered by the integration of a logical (assertional) language and a terminological one. A proposal for such an integration is presented. The assertional language we propose avoids reification and includes a rich set of denotational entities. Our system is shown to be able to handle complex phenomena like periodic events.

1 Introduction.

The development of a system able to engage in a dialog with a user involves considering several representational problems. Representing time is one of these problems, because time permeates our view of reality.

In a financial domain, for example, like that of the consultation system we are developing (the WISBER system, [Bergmann,Gerlach 1987b]) notions like that of duration of an investment or of date, and the associated reasoning processes, are absolutely crucial. The ability to represent and use time is therefore necessary to properly organize both the expert knowledge and the knowledge acquired during the dialog

Being able to store temporal information offers also advantages from the point of view of the organization of the knowledge base. If, for example, the

[1]The project WISBER is sponsored by the German Ministry for Research and Technology.