UNIVERSITA' DEGLI STUDI DI BARI

FACOLTA' DI SCIENZE MATEMATICHE FISICHE E NATURALI
CORSO DI LAUREA IN INFORMATICA
A.A. 2005/06

TESI DI LAUREA IN INGEGNERIA DELLA CONOSCENZA E SISTEMI ESPERTI

THETA-SUSSUNZIONE TRA CLAUSOLE DI HORN: RIDUZIONE DI UN PROBLEMA NP-COMPLETO AD UN PROBLEMA DI SODDISFACIBILITÀ BOOLEANA

Relatore:
Chiar.ma prof.ssa Floriana ESPOSITO

Correlatore:
dott. Nicola DI MAURO

Laureando:
Pasquale, Mauro MINERVINI

Theta-Sussunzione tra clausole di Horn: riduzione di un problema NP-completo ad un problema di Soddisfacibilità Booleana

Pasquale, Mauro Minervini

20 Febbraio 2007

Abstract: l'efficienza delle procedure di dimostrazione nella logica del prim'ordine è un problema rilevante nel caso gli agenti deduttivi vengano usati in ambienti reali, sia da soli sia come componente di un sistema più grande (ad es. in sistemi di apprendimento). Questa tesi propone un nuovo algoritmo di θ-sussunzione basato su una Karp-riduzione polinomiale dell'input in istanze del problema della Soddisfacibilità Booleana.

Indice

1 Introduzione **4**

2 La Programmazione Logica Induttiva **6**
2.1 Introduzione 6
2.2 Induzione 9

3 Il Problema della Theta-Sussunzione **13**
3.1 Introduzione 13
3.1.1 Il problema della Theta-Sussunzione 14
3.1.2 Sussunzione deterministica 15
3.2 Risoluzione basata sui Grafi 16
3.2.1 Riduzione all'isomorfismo tra grafi 16
3.2.2 Riduzione a MAXCLIQUE 19
3.3 Theta-Sussunzione come CSP 21
3.3.1 Procedure di riduzione 23
3.3.2 Procedure di ricerca 25
3.4 FASTheta 25

4 Il Problema della Soddisfacibilità Booleana **29**
4.1 Introduzione 29
4.2 Algoritmi per SAT 30
4.2.1 Algoritmo Davis-Putnam 30
4.2.2 Algoritmo Davis-Logemann-Loveland 32
4.2.3 Algoritmi Stocastici 35

5 Riduzione del Problema della Theta-Sussunzione a SAT **38**
5.1 Spazio di Ricerca 38
5.2 Ridurre lo Spazio di Ricerca 39
5.2.1 Un solo Legame per ogni Variabile 39
5.2.2 Vincoli Strutturali 41
5.3 Soluzione 42

5.4 Esperimenti 43
5.5 Conclusioni 47

Capitolo 1

Introduzione

La Programmazione Logica è un approccio alla programmazione basato sulla rappresentazione dei programmi come teorie della logica del prim'ordine formate da clausole di Horn, la cui esecuzione è ridotta a dimostrare asserzioni in una data teoria. Dato che la classica relazione di dimostrabilità, l'implicazione logica, è indecidibile [SS88], la relazione di generalità (più debole ma decidibile) della θ-sussunzione [Rob65] [Plo70] è usata spesso nella pratica. Date due clausole C e D, C θ-sussume D (scritto come $C \leq_\theta D$) sse esiste una sostituzione θ tale che $C\theta \subseteq D$. Una sostituzione è una associazione da variabili a termini, denotata come $\theta = \{X_1 \to t_1, \cdots, X_n \to t_n\}$, la cui applicazione ad una clausola C, denotata con $C\theta$, riscrive tutte le occorrenze delle variabili $X_i (i = 1, \cdots, n)$ in C con il corrispondente termine t_i.

Nei Dimostratori di Teoremi, l'esplosione delle possibili interazioni tra le clause è spesso limitato eliminando tutte le clausole già sussunte da altre clausole nella teoria. Nella Programmazione Logica Induttiva (ILP), è necessaria una grande quantità di test di sussunzione per verificare la completezza e la consistenza delle nuove ipotesi su tutti gli esempi dati. Un'altro modo di utilizzare la θ-sussunzione consiste nel calcolare la riduzione delle clausole, ossia una clausola che è equivalente ad una data, ma in cui tutti i letterali ridondanti (superflui) sono stati rimossi.

Il problema della Soddisfacibilità Booleana, invece, consiste nel determinare se le variabili di una formula Booleana data possono essere assegnate in modo che la formula venga valutata come $TRUE$: in questo caso, la formula viene detta *soddisfacibile*. La definizione formale del Problema della Soddisfacibilità Booleana (o SAT) richiede che la formula venga espressa in Forma Normale Congiuntiva (o CNF), ossia come congiunzione di disgiunzioni di variabili (o di loro negazioni). In questa forma, ogni disgiunzione

agisce come un vincolo sui possibili valori delle variabili in esso contenute: ad esempio, la clausola $(A \vee \neg B \vee C)$ è soddisfatta per qualsiasi assegnazione ad A, B e C eccetto che $A = C = FALSE$ e $B = TRUE$. Una formula in CNF può essere vista anche come un sistema di vincoli simultanei nello spazio delle assegnazioni di valori di verità sulle variabili in essa contenute. Lo scopo di questa tesi è progettare un algoritmo efficiente per risolvere il problema della θ-sussunzione, sfruttando la predisposizione delle istanze di SAT a esprimere dei vincoli su uno spazio di ricerca e l'efficienza dei moderni SAT solvers.

Capitolo 2

La Programmazione Logica Induttiva

2.1 Introduzione

La Programmazione Logica Induttiva (o ILP, acronimo di Inductive Logic Programming) è un'area di ricerca che ha le sue radici nel Machine Learning e nella Programmazione Logica. Essa ha come scopo quello di creare sia una struttura formale, che degli algoritmi efficienti per l'apprendimento induttivo di descrizioni relazionali nella forma di programmi logici. Dalla programmazione logica la ILP ha ereditato le sue basi teoriche e dal Machine Learning un approccio sperimentale e un orientamento verso le applicazioni pratiche. La ILP ha già dimostrato la sua utilità in diverse aree, tra cui: acquisizione della conoscenza, KDD (Knowledge Discovery in Databases), bioinformatica e NLP (Natural Language Processing).

Formalmente, un problema ILP può essere così definito:

Definizione 1 (problema ILP) *Un problema ILP è una tupla* ($\vdash$, LB, LE, LH) *tale che* $\vdash$ *è una regola di inferenza corretta per un linguaggio del primo ordine L, cioè* $\forall A, B \in L : (A \vdash B \Rightarrow A \models B)$*,* $LB \subseteq L$ *è un linguaggio per rappresentare la conoscenza di base,* $LE \subseteq L$ *è un linguaggio usato per rappresentare gli esempi ed* $LH \subseteq L$ *è un linguaggio per rappresentare le ipotesi. Un algoritmo che accetta qualsiasi* $B \subseteq LB$ *e* $E \subseteq LE$ *come input e calcola un* $H \in LH$ *tale che:*

1. $(B, H, E \nvdash \square)$*, cioè H è consistente con B ed E;*

2. $(B, H \vdash E^+)$*, cioè H e B spiegano* E^+*;*

3. $(\forall e \in E^- : B, H \nvdash e)$*, cioè* H *e* B *non spiegano* E^-*;*

se esiste, o no se non esiste, è chiamato algoritmo di apprendimento ILP per $(\vdash, LB, LE, LH)$*.*

Come si può vedere, la regola di inferenza $\vdash$ ricopre un ruolo centrale nella definizione di un problema ILP. Nei sistemi ILP esistenti, viene utilizzata per diversi motivi. Per cominciare, $\vdash$ è usata per verificare che le ipotesi coprano gli esempi. Diversi sistemi la usano per ridurre la ridondanza all'interno dello spazio delle ipotesi ([Plo70], [MF90]).

Si può notare già da ora come l'efficienza con cui la regola di inferenza $\vdash$ viene implementata, risulti cruciale per l'efficienza degli algoritmi di apprendimento ILP.

Questo capitolo ha lo scopo di fornire una breve introduzione alla ILP.

Il problema dell'Induzione su Clausole di Horn consiste nell'elaborare la definizione di un predicato a partire da un sottoinsieme delle sue istanze ground. Anche se non è impossibile apprendere esclusivamente da esempi positivi, di solito l'esperto fornisce anche esempi negativi per evitare una sovra-generalizzazione. Le definizioni di predicati ausiliari usati nella definizione indotta sono forniti nella forma di teoria di base. Formalmente:

Definizione 2 (termine) *Un termine è una costante, una variabile o una funzione applicata a un qualsiasi insieme di termini.*

Esempio X, c, $f(f(c), f(g(Y)))$ sono termini.

Definizione 3 (letterale) *Un letterale è un qualsiasi predicato (o la sua negazione) applicato a un qualsiasi insieme di termini.*

Esempio $p(c, f(f(c)))$, $p(X)$, $\neg p(X)$ sono letterali.

Definizione 4 (clausola) *Una clausola è una disgiunzione di letterali le cui variabili sono quantificate universalmente. Rappresenteremo la clausola* $\forall V(L_1 \vee \cdots \vee L_n)$ *come l'insieme* $\{L_1, \cdots, L_n\}$*, dove* V *è l'insieme di variabili che occorrono all'interno dei letterali* $L_1, \cdots, L_n$*.*

Esempio $\forall A(r(b) \vee \neg g(A) \vee x)$ è una clausola e verrà rappresentata come $\{r(b), \neg g(A), x\}$.

Definizione 5 (clausola di Horn) *Una clausola di Horn è una clausola che contiene al più un letterale positivo. Una clausola di Horn che ha esattamente un letterale positivo (chiamata clausola definita) può essere scritta nella forma* $H \leftarrow (L_1 \vee \cdots \vee L_n)$*, dove* H *e* $L_1, \cdots, L_n$ *sono letterali positivi:* H *è chiamato testa e* $L_1, \cdots, L_n$ *è chiamato corpo della clausola.*

Esempio Un esempio di clausola di Horn è il seguente: $\neg A \vee \neg B \vee C$.
Il numero dei letterali può essere arbitrario (anche zero); la condizione che al massimo uno sia positivo permette di scrivere la clausola sotto forma di implicazione:
partendo dall'esempio, applichiamo prima De Morgan: $\neg(A \wedge B) \vee C$;
dopodichè utilizziamo l'equivalenza logica: $\neg X \wedge Y \equiv X \Rightarrow Y$;
ricavando quindi: $(A \wedge B) \Rightarrow C$.

Definizione 6 (goal) *Una clausola di Horn priva di letterali positivi viene chiamata clausola goal.*

Esempio $\neg A \vee \neg B \vee \neg C$.

Definizione 7 (fatto) *Una clausola di Horn con corpo vuoto (e quindi senza letterali negativi) viene chiamata fatto e può essere vista come una clausola di Horn definita che si limita ad asserire una determinata proposizione.*

Esempio $padre(marco, anna)$.

Definizione 8 (clausola ground) *Una clausola priva di variabili viene definita ground.*

Esempio $x \vee y \wedge \neg z$.

Definizione 9 (Problema dell'Induzione su Clausole di Horn) *Il Problema dell'Induzione su Clausole di Horn può essere definito come segue:*
Dati:

1. *P: fatti ground da essere richiesti (esempi positivi);*

2. *N: fatti ground da non essere richiesti (esempi negativi);*

3. *B: un insieme di definizioni di predicato (teoria di base);*

4. Γ*: il linguaggio delle ipotesi;*

Trova *una definizione di un predicato* $H \in \Gamma$ *(ipotesi) tale che:*

1. $\forall p \in P : B \cup H \models p$ *(completezza);*

2. $\forall n \in N : B \cup H \not\models n$ *(consistenza);*

Per esempio, una istanza di questo problema potrebbe essere l'apprendere una definizione per il predicato $r/3$ dai seguenti esempi (considerando una teoria di base vuota):
Esempi positivi:

1. r([2, 1], [3], [1, 2, 3]).
2. r([a], [], [a]).
3. r([1], [2, 3], [1, 2, 3]).
4. r([], [a], [a]).
5. r([], [], []).

Esempi negativi:

1. r([1, 2], [3], [1, 2, 3]).
2. r([a], [a], [a]).
3. r([], [a], [b]).

In generale, esistono due tipi di approccio al problema dell'induzione su clausole di Horn: è possibile partire

1. dalle clausole più corte, aggiungendo letterali ai loro corpi in maniera progressiva man mano che risultano essere eccessivamente generali (approccio *top-down*);
2. da quelle più lunghe, rimuovendo letterali dai loro corpi in maniera progressiva fin quando non diventano eccessivamente generali (approccio *bottom-up*).

2.2 Induzione

L'induzione top-down è un approccio di tipo generate-then-test: le clausole di ipotesi sono generate in un ordine predeterminato e quindi testate sugli esempi.
Un possibile funzionamento di un sistema di questo tipo potrebbe essere il seguente: il sistema si inizializza con la definizione più generale. Per ogni

esempio negativo, le clausole che descrivono il modello vengono *specializzate* in modo che coprano tutti gli esempi positivi e non quelli negativi.

Le clausole possono essere specializzate in due modi: applicando una sostituzione o aggiungendo un letterale al corpo. Ciò può essere formalizzato come segue:

Definizione 10 (sostituzione) *Una sostituzione è una qualsiasi funzione che sostituisce le variabili con termini. Data una sostituzione θ ed un letterale L, scriviamo $L\theta$ per denotare il risultato dell'applicazione della sostituzione θ ad L. Una sostituzione è rappresentata come un insieme $\{V_1 \rightarrow T_1, \cdots, V_n \rightarrow T_n\}$, dove $V_1, \cdots, V_n$ sono variabili, $T_1, \cdots, T_n$ sono termini e $X \rightarrow Y$ significa che la variabile X va sostituita con il termine Y. Possiamo applicare una sostituzione θ ad una clausola C, scritto $C\theta$, applicando la sostituzione θ ad ogni letterale in C.*

Esempio Sia C la clausola $\{f(a), X, G(Y,a)\}$ e θ la sostituzione $\{X \rightarrow f(x), Y \rightarrow b\}$; il risultato dell'applicazione di θ a C è $C\theta = \{f(a), f(x), G(b,a)\}$.

Definizione 11 (θ-sussunzione) *Una clausola C θ-sussume una clausola D, scritto $C \leq_\theta D$, sse esiste una sostituzione θ tale che $C\theta \subseteq D$. Se C θ-sussume D, allora si dice che C generalizza D.*

Esempio Sia C la clausola $\{f(X,d), g(Y)\}$ e D la clausola $\{f(d,d), G(A), p(t)\}$; C θ-sussume D con la sostituzione $\theta = \{X \rightarrow d, Y \rightarrow A\}$.

La θ-sussunzione è riflessiva e transitiva, ma non antisimmetrica (ossia $p(X) : -q(X)$ e $p(X) : -q(X), q(Y)$ si θ-sussumono tra loro): in questo modo viene definito un preordinamento sull'insieme delle clausole, cioè un insieme di classi di equivalenza parzialmente ordinato. Se definiamo una clausola da ridurre, se essa non θ-sussume nessuna delle sue sottoclausole, allora ogni classe di equivalenza contiene una clausola ridotta che è unica a meno di una ridenominazione delle variabili. L'insieme di queste classi di equivalenza formano un *reticolo*:

Definizione 12 (reticolo) *Un reticolo è un insieme parzialmente ordinato in cui tutti i sottoinsiemi limitati non vuoti ammettono sia un estremo inferiore (inf) sia un estremo superiore (sup).*

Ossia, due clausole hanno un unico minimo limite superiore ed un unico massimo limite inferiore sotto θ-sussunzione. Ci riferiremo al minimo limite superiore di due clausole rispetto alla θ-sussunzione come al loro θLGG (Least General Generalisation, generalizzazione meno generale, rispetto ad una sostituzione θ). Formalmente:

Definizione 13 (θLGG) *Una clausola C è un θLGG di un insieme di clausole S se:*

1. *$\forall c \in S : C \leq_\theta E$ (C generalizza ogni clausola in S);*
2. *($\exists D$ tale che $\forall c \in S, D \leq_\theta c$) $\Rightarrow D \leq_\theta C$ (C è la più piccola clausola che soddisfa il punto 1);*

Da notare che il reticolo non contiene catene discendenti infinite.

Chiaramente, se C_1 θ-sussume C_2 allora C_1 richiede C_2, ma non è vero il contrario.

Sembra che l'ordinamento creato dall'implicazione logica sia quello da usare, in particolare quando si vogliono apprendere clausole ricorsive. Sfortunatamente, il minimo limite superiore di due clausole di Horn con l'implicazione logica non è definito. Il motivo è semplicemente che, in generale, questo minimo limite superiore sarebbe dato dalla disgiunzione delle due clausole, ma questo potrebbe non essere una clausola di Horn. Inoltre, le generalizzazioni con l'implicazione logica non sono facilmente calcolabili, considerando che generalizzazione e specializzazione usando la θ-sussunzione sono semplici operazioni sintattiche. Infine, c'è da considerare che l'implicazione tra clausole è indecidibile [SS88], considerando che la θ-sussunzione è decidibile (anche se *NP*-completa). *Per queste ragioni, i sistemi ILP di solito impiegano la θ-sussunzione anzichè l'implicazione logica.*

Mentre gli approcci top-down specializzano passo per passo una clausola iniziale molto generale, gli approcci bottom-up generalizzano clausole definite molto specifiche.
Un metodo per eseguire questo compito è il seguente: si calcola il θLGG delle clausole ground, che può essere ottenuto usando l'*anti-unificazione*, l'opposto della unificazione, che confronta i sottotermini nella stessa posizione e li trasforma in variabili se differiscono. Uno dei principali problemi, in questo caso, è selezionare il letterale ground giusto da un insieme molto più ampio. Supponiamo di sapere quali letterali scegliere dalla testa ma non dal corpo: un approccio consiste nel raggruppare tutti i letterali nei corpi di entrambe le clausole ground. Dato che i corpi delle clausole sono, dal punto di vista logico, disordinati, il θLGG è ottenuto anti-unificando tutte le possibili coppie di letterali nei corpi, tenendo presenti le variabili che sono state introdotte quando sono state anti-unificate le teste. Dopo aver costruito tale clausola, il nostro obiettivo consiste nel generalizzarla eliminando il maggior numero di letterali possibile (ad esempio, possiamo rimuovere i letterali ground, dato che corrispondono ai nostri esempi originali). In sostanza, è ragionevole

richiedere che la clausola sia *connessa*, ossia che ogni letterale nel corpo condivida una variabile con la testa o con un altro letterale appartenente al corpo che sia connesso con la testa. Inoltre, si potrebbe fare uso degli esempi negativi per verificare che la clausola non sia troppo generale, se sono stati rimossi alcuni letterali dal corpo.

Capitolo 3

Il Problema della Theta-Sussunzione

3.1 Introduzione

Come abbiamo visto, il problema della θ-sussunzione è cruciale per l'efficienza dei sistemi di apprendimento che fanno uso dell'ILP. Questo capitolo ha lo scopo di illustrare i principali algoritmi di θ-sussunzione sviluppati fin'ora.
La θ-sussunzione è un tipo di *relazione di conseguenza* con le caratteristiche di essere *decidibile* e *corretta* (ma *incompleta*), mentre l'implicazione logica è, in generale, *indecidibile*. Una clausola C θ-sussume D, scritto $C \leq_\theta D$, sse esiste una sostituzione θ tale che $C\theta \subseteq D$.
La θ-sussunzione viene usata come relazione di conseguenza in vari sistemi ILP per stabilire se una regola copre un esempio ([MF90], [vdLNC93], [DD97]: specialmente il test di consistenza, cioè il test che stabilisce se una nuova clausola generalizzata copre gli esempi negativi, richiede un gran numero di test di sussunzione. Quindi, algoritmi di sussunzione efficienti che non dipendono dalle restrizioni imposte dal linguaggio delle ipotesi rappresentano un importante contributo all'ILP.
Il problema della θ-sussunzione è *NP*-completo nel caso generale [KN86]: la *NP*-completezza deriva dall'ambiguità dell'identificazione delle variabili.
Se, in una clausola, può essere trovato un letterale che corrisponde esattamente ad un letterale dell'altra clausola, allora il letterale può essere corrisposto in modo *deterministico* e non vi è bisogno di eseguire alcun tipo di backtracking. Però la complessità cresce esponenzialmente col numero dei letterali non-deterministici rimanenti.

3.1.1 Il problema della Theta-Sussunzione

La θ-sussunzione è una approssimazione dell'implicazione logica. Una clausola C θ-sussume una clausola D, scritto $C \leq_\theta D$, sse esiste una sostituzione θ tale che $C\theta \subseteq D$ e $|C| \leq |D|$. Useremo il termine sussunzione al posto di θ-sussunzione.
Mentre l'implicazione, in generale, non è decidibile per i linguaggi del prim'ordine, la θ-sussunzione è decidibile ma incompleta, ossia possono esistere due clausole C e D tali che $C \nvdash_\theta D$ ma $C \vDash D$: questo succede se C è auto-risolvente (ricorsiva) o se D è tautologica [Got87]; se escludiamo le clausole tautologiche e auto-risolventi, allora $C \vdash D \Leftrightarrow C \vDash D$ [Got87] [Mug92] [KL94].
Il problema della θ-sussunzione è NP-completo in generale. La complessità nel caso pessimo è $O(vars(D)^{vars(C)})$, oppure $O(|D|^{|C|})$.

Nella teoria della complessità, i problemi NP-completi sono i problemi più *difficili* in NP (classe dei problemi risolvibili da Macchine di Turing non deterministiche in tempo polinomiale), nel senso che rappresentano la più piccola sottoclasse di NP che potrebbe, in teoria, rimanere fuori da P (classe dei problemi risolvibili da Macchine di Turing deterministiche in tempo polinomiale). Il motivo è che una soluzione algoritmica deterministica e con complessità polinomiale a qualsiasi problema NP-completo sarebbe anche una soluzione a qualsiasi altro problema in NP.
Formalmente:

Definizione 14 (Karp-riduzione) *Un problema di decisione P_1 è Karp-riducibile (o riducibile molti-a-uno) a un problema di decisione P_2 (denotato con $P_1 \leq_m P_2$) sse esiste un algoritmo R che trasforma ogni istanza $x \in I_{P_1}$ di P_1 in una istanza $y \in I_{P_2}$ di P_2 in modo tale che $x \in Y_{P_1}$ sse $y \in Y_{P_2}$. R è detta Karp-riduzione da P_1 a P_2.*

Se esiste una Karp-riduzione R da P_1 a P_2 e se si conosce un algoritmo A_2 per P_2, allora si può ottenere un algoritmo A_1 per P_1 così:
data una istanza $x \in I_{P_1}$, si applica R a x e si ottiene $y \in I_{P_2}$;
si applica A_2 a y: se A_2 restituisce VERO allora si restituisce VERO, altrimenti si restituisce FALSO.

La complessità in tempo di A_1 è data dalla somma delle complessità in tempo di R e A_2.

Definizione 15 (problema NP-completo) *un problema di decisione C viene definito NP-completo se è completo per NP, ossia:*

1. $C \in NP$

2. *è NP-hard, cioè ogni altro problema in NP è Karp-riducibile ad esso.*

Per dimostrare che un problema A in *NP* è *NP*-completo è sufficiente mostrare che un altro problema che si sa essere *NP*-completo può essere ridotto ad A.
Allo stato attuale, tutti gli algoritmi conosciuti che risolvono problemi *NP*-completi hanno una complessità superpolinomiale rispetto alla dimensione dell'input. Non si conoscono algoritmi più veloci, e gli algoritmi che risolvono il problema della θ-sussunzione non fanno eccezione (nel caso generale).

Verrà definito ora un classico problema *NP*-completo che sarà utile nella dimostrazione:

Definizione 16 (3-Soddisfacibilità) *(o 3SAT) è un problema ottenuto dal problema NP-completo Soddisfacibilità, considerando nella sua definizione soltanto le clausole aventi 3 termini:*
ISTANZA*: una formula in forma normale congiuntiva S su un insieme V di variabili booleane, con ogni clausola composta da esattamente 3 termini.*
PREDICATO*: esiste una assegnazione $f : X \mapsto \{VERO, FALSO\}$ che soddisfa F?*

Una dimostrazione della *NP*-completezza del problema della θ-sussunzione tramite una riduzione a 3SAT può essere trovata su [Bax76] e [Bax77].

3.1.2 Sussunzione deterministica

Un modo per far fronte alla *NP*-completezza della θ-sussunzione è la *sussunzione deterministica* [DMR92] [KL94]. Una clausola viene detta *determinata* se per ogni letterale esiste esattamente un possibile abbinamento consistente con i letterali precedentemente abbinati o, più in generale, se esiste un ordinamento dei letterali tale che ad ogni passo, per ogni letterale, esiste esattamente un abbinamento consistente con i letterali precedentemente abbinati.

Definizione 17 (sussunzione deterministica) *Siano $C = c_0 \leftarrow \{c_i\}$ $D = d_0 \leftarrow d_1, \cdots, d_m$ due clausole di Horn. C θ-sussume D in modo deterministico, scritto $C \leq_{\theta DET} D$ con $\theta = \theta_0\theta_1 \cdots \theta_n$, sse esiste un ordinamento $c_1, \cdots, c_n$ dei c_i tale che $\forall i, 1 \leq i \leq n, \exists_1 \theta_i : \{c_1, \cdots, c_i\}\theta_0 \cdots \theta_i \subseteq D$.*

$C \leq_{\theta DET} D$ può essere verificata con al più $O(|C|^2 * |D|)$ tentativi di unificazione dal seguente algoritmo:
Il problema principale di quest'approccio è che le condizioni sono molto restrittive.

1. Se esiste un letterale $l_1 \in C$ a cui corrisponde esattamente un letterale $l_2 \in D$ tale che $l_1\mu = l_2$, sostituisci C con μ.
2. Se esiste un letterale in C a cui non corrisponde alcun letterale in D, allora $C\theta \nsubseteq D$.
3. Se un qualsiasi letterale non può essere abbinato in modo unico, inizia con la clausola sostiuita fin'ora e verifica la sussunzione usando un algoritmo di backtracking.

3.2 Risoluzione basata sui Grafi

Definizione 18 (grafo) *Un grafo G è una coppia ordinata $G = (V, A)$ in cui:*

1. *V è un insieme di vertici (anche chiamati nodi);*
2. *A è un insieme di coppie ordinate (v_i, v_j) tali che $v_i, v_j \in V$ (ossia $A \subseteq V \times V$);*

Definizione 19 (clique di un grafo) *Un insieme di nodi $C \subseteq V$ viene definito clique di un grafo (V, A) sse $C \times C \subseteq A$, ossia tutti i nodi sono mutualmente adiacenti.*

Definizione 20 (isomorfismo tra grafi) *Siano G e H due grafi aventi vertici $V(G) = \{u_1, u_2, \cdots, u_n\}$ e $V(H) = \{v_1, v_2, \cdots, v_m\}$ e archi $A(G) \subseteq V(G) \times V(G)$ e $A(H) \subseteq V(H) \times V(H)$. Un isomorfismo tra grafi è una bigezione $f : V(G) \rightarrow V(H)$ tale che $(u, v) \in A(G)$ sse $(f(u), f(v)) \in A(H)$. Se esiste un isomorfismo da G a H, allora G viene definito come isomorfo a H, scritto $G \cong H$.*

Definizione 21 (sottografo) *Un grafo $H = (V_H, A_H)$ viene definito sottografo di $G = (V, A)$ sse $V_H \subseteq V$ e $A_H \subseteq A \times A$ (con $A_H \subseteq V_H \times V_H$). Se H è un sottografo di G, G viene definito supergrafo di H.*

3.2.1 Riduzione all'isomorfismo tra grafi

In [WKS81] viene proposto un metodo per risolvere il problema dell'isomorfismo tra grafi riducendo il numero dei possibili accoppiamenti usando l'informazione sul contesto. In [SHW96] viene ridotto il problema della sussunzione all'isomorfismo tra grafi definendo il *grafo di occorrenza di una clausola*:

Definizione 22 (clausola Datalog) *Una clausola Datalog è una clausola in cui tutti i letterali hanno parametri ristretti a variabili o costanti (ossia in una clausola Datalog non esistono funzioni).*

Esempio $\{r(A, b), \not f(x)\}$ è una clausola Datalog, mentre $\{f(x, s(x)), a\}$ non è una clausola Datalog dato che il secondo argomento del letterale $f(x, s(x))$ è una funzione di x.

Definizione 23 (grafo di occorrenza) $C, A_C)$ *è il grafo di occorrenza di una clausola Datalog* C *se* $(l_i, l_j, \pi_i \leftrightarrow \pi_j) \in A_C$ *sse esiste una cariabile* x *che occorre nel letterale* l_i *come* π_i*-esimo argomento e nel letterale* l_j *come* π_j*-esimo argomento. Gli archi del grafo di occorrenza sono marcati* $\pi_i \leftrightarrow \pi_j$*, dove* π_i *e* π_j *sono le posizioni degli argomenti in cui sono trovate le occorrenze di una stessa variabile.*

Esempio Sia $C = \{p(a, b), p(b, c), q(x, y, c)\}$ una clausola Datalog. Il grafo di occorrenza di C ha un arco $(p(a, b), p(b, c), 2 \leftrightarrow 1)$, dato che i letterali $p(a, b)$ e $p(b, c)$ hanno la variabile b in posizione, rispettivamente, 2 ed 1. In modo simile, possiamo osservare che C ha un arco $(p(b, c), q(x, y, c), 2 \leftrightarrow 3)$, dato che c si presenta in $p(b, c)$ in posizione 2 e in $q(x, y, c)$ in posizione 3.

Il contesto relativo a un grafo con profondità d di un letterale è l'insieme di tutti i percorsi di lunghezza d che partono da quel letterale nel grafo di occorrenza.

Definizione 24 (contesto relativo a un grafo) *Il contesto relativo a un grafo di profondità* d *di un letterale* l_1 *appartenente ad una clausola* C*, denotato con* $con_{gra}(l_1, d, C)$*, è l'insieme di termini* $p_1 \cdot \pi_1 \leftrightarrow \pi_2 \cdot p_2 \cdot \cdots \cdot \pi_{n-1} \leftrightarrow \pi_d \cdot p_d$*, sse esiste un insieme* $\{(l_1, l_2, \pi_1 \leftrightarrow \pi_2), \cdots, (l_{d-1}, l_d, \pi_{d-1} \leftrightarrow \pi_d)\}$ *di archi nel grafo di occorrenza, tale che* p_i *è il simbolo di predicato di* l_i *(è anche possibile che* $l_1 = l_d$*)*

Esempio Sia $C = \{p(a, b), p(b, c), q(x, y, c)\}$ una clausola Datalog. Il grafo di occorrenza di questa clausola ha, come abbiamo visto nel precedente esempio, archi $\{(p(a, b), p(b, c), 2 \leftrightarrow 1), (p(b, c), q(x, y, c), 2 \leftrightarrow 3)\}$. Il contesto del letterale $p(a, b)$ a profondità 1 è $con_{gra}(p(a, b), 1, C) = \{p \cdot 2 \leftrightarrow 1 \cdot p\}$ ed a profondità 2 è $con_{gra}(p(a, b), 2, C) = \{p \cdot 2 \leftrightarrow 1 \cdot p \cdot 2 \leftrightarrow 3 \cdot q, p \cdot 2 \leftrightarrow 1 \cdot p \cdot 1 \leftrightarrow 2 \cdot p\}$. Il primo cammino di $con_{gra}(p(a, b), 2, C)$ corrisponde alla sequenza di letterali $\langle p(a, b), p(b, c), q(x, y, c)\rangle$, ed il secondo alla sequenza $\langle p(a, b), p(b, c), p(a, b)\rangle$.

Teorema 1 *Siano* C *e* D *clausole e* $l_1 \in C$*,* $l_2 \in D$ *letterali. Sia* $d \in \mathbb{N}$ *la profondità e sia* $l_1\mu = l_2$*, dove* μ *è una sostituzione. Se* $con_{gra}(l_1, d, C) \not\subseteq con_{gra}(l_2, d, D)$*, allora non esiste nessun* θ *tale che* $C\mu\theta \subseteq D$*.*

Dimostrazione Siano C e D due clausole e siano $con_{gra}(l_1, d, C)$ e $con_{gra}(l_2, d, D)$, rispettivamente, i contesti di un letterale di C e di un letterale di D tali che $con_{gra}(l_1, d, C) \nsubseteq con_{gra}(l_2, d, D)$. Ciò implica che esiste una sequenza di letterali di C che condividono almeno una variabile con i loro vicini che non ha alcuna corrispondenza in D. Sia $p_\alpha \cdot \pi_\alpha \leftrightarrow \pi_\beta \cdot p_\beta$ la parte critica del cammino, e l_α, l_β la coppia di letterali senza alcun cammino corrispondente in D. Se esiste un θ tale che $C\theta \subseteq D$, allora $\pi_\alpha(l_\alpha\theta = \pi_\beta(l_\beta)\theta$, dove π seleziona l'argomento, dato che le variabili a queste posizioni sono uguali. Ma $l_\alpha\theta$ e $l_\beta\theta$ non possono essere elementi di D, perchè condividono una variabile comune e abbiamo assunto che non ci fosse alcun cammino corrispondente in D.

Da ciò deduciamo che un letterale non può essere abbinato ad un altro letterale se il suo contesto non può essere incluso in quello dell'altro letterale. Ciò implica che un letterale può essere abbinato ad un altro letterale solo se il suo contesto è un sottoinsieme del contesto dell'altro letterale. Questo dà la possibilità di ottenere il seguente algoritmo, che è una estensione della sussunzione deterministica:

Require:

for all Per ogni letterale $l_1 \in C$ **do**

Se $con_{gra}(l_1, C, d)$ è un sottoinsieme del contesto relativo al grafo costituito esattamente da un letterale $l_2 \in D$ con $l_1\mu = l_2$, sostituisci C con μ;

Se il grafo relativo al contesto di l_1 non può essere incluso nel grafo relativo al contesto di un letterale $l_2 \in D$, allora C non può sussumere D.

Se l_1 ed il grafo relativo al suo contesto corrisponde a dei letterali in D in modo non deterministico, aggiungi l_1 a C'.

end for

Inizia con la clausola C' sostituita fin'ora e usa un algoritmo di backtracking per verificare se $C'\theta \subseteq D$. Usa il grafo relativo al contesto per ridurre il numero dei letterali candidati in D per ogni letterale in C'.

Esempio Consideriamo le clausole Datalog $C = \{p(a, b), p(b, c), q(x, y, c)\}$ e $D = \{p(r, s), p(s, t), q(p, q, t)\}$. Il grafo di occorrenza di C ha archi $\{(p(a, b), p(b, c), 2 \leftrightarrow 1), (p(b, c), q(x, y, c), 2 \leftrightarrow 3)\}$, mentre quello di D ha archi $\{(p(r, s), p(s, t), 2 \leftrightarrow 1), (p(s, t), q(p, q, t), 2 \leftrightarrow 3)\}$. Il contesto di profondità 1 di $p(a, b)$ è $con_{gra}(p(a, b), 1, C) = \{p\cdot 2 \leftrightarrow 1\cdot p\}$. In modo simile, $con_{gra}(p(b, c), 1, C) = \{p \cdot 1 \leftrightarrow 2 \cdot p, p \cdot 2 \leftrightarrow 3 \cdot q\}$ è un sottoinsieme di $con_{gra}(p(s, t), 1, D) = \{p \cdot 1 \leftrightarrow 2 \cdot p, p \cdot 2 \leftrightarrow 3 \cdot q\}$ e $con_{gra}(q(x, y, c), 1, C) = \{q \cdot 3 \leftrightarrow 2 \cdot p\}$ è un sottoinsieme di $con_{gra}(q(p, q, t), 1, D) = \{q \cdot 3 \leftrightarrow 2 \cdot p\}$. Ogni letterale in C corrisponde ad un letterale in D ed il contesto di quel letterale in C è un

sottoinsieme di quello del corrispondente letterale in D, quindi C sussume D.

3.2.2 Riduzione a MAXCLIQUE

Un'ulteriore strategia consiste nel ridurre il problema della sussunzione al problema del trovare il massimo *clique* di un grafo di sostituzione. Ogni vertice nel grafo di sostituzione, per verificare se $C\theta \subseteq D$, è una sostituzione che abbina alcuni letterali in C ad uno in D.
Formalmente:

Definizione 25 (clique) *Un insieme di nodi $C \subseteq V$ è un clique di un grafo $G = (V, A)$ sse $C \times C \subseteq A$. In altre parole, un clique è un insieme di nodi mutualmente adiacenti in un grafo.*

Definizione 26 (compatibilità forte) *due sostituzioni θ_1 e θ_2 sono dette fortemente compatibili sse $\theta_1 \cdot \theta_2 = \theta_2 \cdot \theta_1$. Ciò implica che nessuna variabile è riassegnate in θ_1 o θ_2.*

Esempio Le sostituzioni $\theta_1 = \{X/a\}$ e $\theta_2 = \{X/b\}$ non sono fortemente compatibili perchè è chiaro che l'ordine delle sostituzioni influenzerà il valore con cui sarà rimpiazzata X. In modo simile, $\theta_1 = \{X/A\}$, $\theta_2 = \{A/b\}$ non sono fortemente compatibili dato che $X \cdot \theta_1 \cdot \theta_2 = b$ mentre $X \cdot \theta_2 \cdot \theta_1 = A$.

Definizione 27 () *$uni(C, l_i, D) = \{\mu \mid l_i \in C, l_i\mu \in D\}$ è l'insieme di tutte le sostituzioni che trasformano un letterale l_i della clausola C in un qualche letterale di D.*

Esempio Siano $C = \{p(X, Y)\}$ e $D = \{p(a, b), p(c, d)\}$ due clausole, allora $uni(C, p(X, Y), D) = \{\{X/a, Y/b\}, \{X/c, Y/d\}\}$.

Se troviamo il prodotto Cartesiano dell'*uni* di tutti i letterali in C, allora se $C\theta \subseteq D$, un elemento di questo prodotto Cartesiano deve essere θ.

Proposizione 1 (Eisinger) *Una clausola C sussume una clausola D, con $C\theta \subseteq D$, sse esiste una n-tupla $(\theta_1, \cdots, \theta_n) \in \times_{i=1}^{n} uni(C, l_i, D)$, dove $n = |C|$, tale che tutte le θ_i sono, a coppie, fortemente compatibili.*

Esempio Siano $C = \{P(x, y), P(y, z)\}$ e $D = \{P(a, b), P(b, c), Q(d)\}$. Allora $uni(C, P(x, y), D) = \{\{x \to a, y \to b\}, \{x \to b, y \to c\}\}$ e $uni(C, P(y, z), D) = \{\{y \to a, z \to b\}, \{y \to b, z \to c\}\}$. Il prodotto cartesiano di questi insiemi è $\times_{i=1}^{2} uni(C, l_i, D) = \{\{x \to a, y \to b\} \cdot \{y \to a, z \to b\}, \{x \to a, y \to b\} \cdot \{y \to b, z \to c\}, \{x \to b, y \to c\} \cdot \{y \to a, z \to b\}, \{x \to b, y \to c\} \cdot \{y \to b, z \to c\}\}$, di cui solo $\{x \to a, y \to b\} \cdot \{y \to b, z \to c\} = \{x \to a, y \to b, z \to c\}$ è una sostituzione fortemente compatibile.

Se siamo alla ricerca di un θ valido, tale che $C\theta \subseteq D$, allora nel caso pessimo avremo bisogno di enumerare $|D|^{|C|}$ sostituzioni: questo perchè la dimensione di $uni(C, l_i, D)$ per dei letterali $l_i \in C$ è al più $|D|$. Per la proposizione precedente, θ deve essere un elemento dell'insieme $\times_{i=1}^{n} uni(C, l_i, D)$ che ha dimensioni $|D|^{|C|}$, dato che $n = |C|$.

Scheffer etc. riducono la sussunzione al problema di trovare il massimo clique definendo un grafo di sostituzione. I vertici del grafo sono insiemi di sostituzioni da un qualsiasi letterale in C a un qualche letterale in D. Un arco esiste in questo grafo se due sostituzioni sono fortemente compatibili.

Definizione 28 (grafo di sostituzione) *Siano C e D due clausole e $n = |C|$. G è il grafo di sostituzione di C e D sse $V(G) = \bigcup_{i=1}^{n}(uni(C, l_i, D), i)$ e $((\theta_1, i), ((\theta_2, j)) \in A(G)$ sse θ_1 e θ_2 sono fortemente compatibili ed $i \neq j$.*

Proposizione 2 *Siano C e D due clausole. Allora $C\theta \subseteq D$ con $\theta = \theta_1 \cdots \theta_n$ sse esiste un clique $\{\theta_1, \cdots, \theta_n\}$ di cardinalità $|C|$ nel grafo di sostituzione di C e D.*

Esempio Sia C la clausola $\{p(A, B),\ p(Q, B)\}$ e D la clausola $\{p(x, y), p(y, z)\}$. L'insieme dei vertici nel grafo di sostituzione è $\{\{A/x, B/y\}, \{A/y, B/z\}, \{Q/x, B/y\}, \{Q/y, B/z\}\}$ di cui le coppie $\{(\{A/x, B/y\}, \{Q/x, B/y\}), (\{A/y, B/z\}, \{Q/y, B/z\})\}$ sono fortemente compatibili e formano archi nel grafo di sostituzione. Il massimo clique di questo grafo ha grandezza $|C| = 2$, per cui C sussume D per la precedente proposizione.

Il seguente algoritmo è una semplice procedura per verificare se una clausola C sussume una clausola D usando il metodo del massimo clique:

---

1. Crea il grafo di sostituzione S delle clausole C e D.
2. Calcola il massimo clique, M, di S:
   (a) se $|M| = |C|$ allora C sussume D.
   (b) se $|M| < |C|$ allora C non sussume D.

Carraghan e Pardalos hanno proposto il seguente algoritmo per determinare il più grande clique di un grafo [CP90]:

1. Inizia con gli insiemi $Nodi$, contenente tutti i nodi, $Clique$, inizialmente vuoto, ed una ricorsione $profondità$, inizialmente ad 1.

2. Per ogni nodo $\theta_i \in Nodi$ ripeti:

   (a) crea un nuovo insieme $Nodi' = Nodi \cap adiacenti(\theta_i)$;

   (b) se $|Clique| < profondità + |Nodi'|$ (e $Nodi' \neq \emptyset$) allora ricomincia ricorsivamente dal punto (2) con $profondità$+1 e $Nodi'$, determina il massimo clique di $Nodi'$.

   (c) se $|\{\theta_i \cup massimoClique(Nodi')| > Clique$, salva il nuovo clique in $Clique$.

   (d) rimuovi θ_i e continua il ciclo dal punto (2).

3. Restituisci $Clique$.

3.3 Theta-Sussunzione come CSP

Questa sezione serve da introduzione ai Problemi di Soddisfacimento dei Vincoli (CSP, da Constraint Satisfaction Problem) e presenta alcuni algoritmi di risoluzione.
Un CSP è un problema composto da un insieme finito di variabili, di cui ciascuna è associata ad un dominio finito, ed un insieme di vincoli che restringono i valori che le variabili possono assumere allo stesso tempo. Il problema viene risolto quando viene assegnato un valore a ciascuna variabile in modo tale che tutti i vincoli siano soddisfatti.
Formalmente:

Definizione 29 (problema di Soddisfacimento dei Vincoli (CSP)) *Un CSP è costituito da:*

1. *Un insieme di n variabili $\chi = \{X_1, \cdots, X_n\}$; ad ogni variabile X_i è associata la sua variabile dominio $dom(X_i)$ che include tutti i possibili valori di X_i.*

2. *Un insieme di m vincoli, che specificano i valori simultaneamente ammissibili delle variabili.*

Ogni vincolo r è definito su alcune variabili di χ:

1. *Un vincolo può essere pensato come un predicato k-ario $r(X_{i1}, \cdots, X_{ik})$.*

2. *L'ambito di un vincolo, denotato con* $arg(r)$*, è l'insieme di variabili* $\{X_{i1}, \cdots, X_{ik}\}$.

3. *L'insieme dei valori ammissibili delle variabili nell'ambito di un vincolo viene chiamato relazione vincolata, denotata con* $rel(r)$*: può essere considerato come un insieme di* N *letterali* $\{r(a_{1_{i1}}, \cdots, r(a_{1_{ik}}), \cdots, r(a_{N_{i1}}, \cdots, r(a_{N_{ik}})\}$*, con* $a_{l_{ij}}$ *un valore in* $dom(X_{i_j})$*, con* $l = 1, \cdots, N$ *e* $j = 1, \cdots, k$.

Nonostante una relazione vincolata possa essere infinita nel caso generale, nel resto del documento verranno considerate esclusivamente relazioni vincolate finite.

Definizione 30 (CSP Soddisfacibile) *Un CSP viene definito soddisfacibile sse ammette una soluzione, ossia una assegnazione ad ogni variabile* X_i *di un valore* $a_i \in dom(X_i)$ *in modo tale che tutti i vincoli sono soddisfatti. Una soluzione può essere vista come un abbinamento* $\theta = \{X_i/a_i\}$ *tale che per ogni vincolo* $r(X_{i1}, \cdots, X_{ik})$ *risulta che* $r\theta = r(a_{i1}, \cdots, a_{ik}) \in rel(r)$*. In altre parole, il CSP definito dai vincoli* $r_1, \cdots, r_m$ *è soddisfacibile sse* $r_1 \cdot \cdots \cdot r_m$ θ*-sussume la congiunzione di tutti i letterali in* $rel(r_1), \cdots, rel(r_m)$.

Esempio Consideriamo un CSP definito nel seguente modo:

1. $\chi = \{X_0, X_1, X_2, X_3\}$, dove ogni variabile ha dominio $\{m, m_1, m_2\}$;

2. I 6 vincoli sono definiti come segue:

   (a) tc è definito su X_0 con $rel(tc) = m$;

   (b) atm_1 è definito su (X_0, X_1) con dominio $\{\langle m, m_1\rangle, \langle m, m_2\rangle\}$;

   (c) atm_2 è definito su (X_0, X_2) con dominio $\{\langle m, m_1\rangle, \langle m, m_2\rangle\}$;

   (d) atm_3 è definito su (X_0, X_3) con dominio $\{\langle m, m_1\rangle, \langle m, m_2\rangle\}$;

   (e) $bond_1$ è definito su (X_1, X_2) con dominio $\{\langle m_1, m_2\rangle\}$;

   (f) $bond_2$ è definito su (X_1, X_3) con dominio $\{\langle m_1, m_2\rangle\}$;

Questo CSP è equivalente a verificare se C θ-sussume D, dove C e D sono definite come segue:

1. C: $tc(X_0) : -atm(X_0, X_1)$, $atm(X_0, X_2)$, $atm(X_0, X_3)$, $bond(X_1, X_2)$, $bond(X_1, X_3)$.

2. D: $tc(m) : -atm(m, m_1), atm(m, m_2), bond(m_1, m_2)$.

Due CSP sono *equivalenti* sse sono definiti sulle stesse variabili e ammettono le stesse soluzioni. Dato che qualsiasi CSP può essere incluso in un CSP binario, ossia che considera solo vincoli binari, la maggior parte degli algoritmi che risolvono questa tipologia di problemi considerano solo CSP binari. Nel resto del documento, assumeremo che non c'è perdita di generalità e che esiste al più un vincolo per ogni coppia di variabili.

Complessità

La complessità di un CSP è esponenziale nel numero n delle variabili e lineare nel numero m dei vincoli. Più precisamente, assumendo che tutte le variabili abbiano un dominio della stessa dimensione L, la complessità della risoluzione di un CSP è $O(L^n \times m)$.
Un primo metodo per diminuire la complessità è decomporre il CSP in sottoproblemi indipendenti o scarsamente correlati, gerarchizzando l'insieme delle variabili o l'insieme dei vincoli.
Gli algoritmi che risolvono CSP fondamentalmente combinano due tipologie di procedure: procedure di riduzione e procedure di ricerca: le procedure di riduzione riducono i domini delle variabili, con il risultato di trasformare un CSP in uno equivalente con una complessità minore, mentre le procedure di ricerca dipendono dalla strategia di ricerca e dalle caratteristiche dello *spazio di ricerca* (l'insieme di tutte le soluzioni candidate del problema).

3.3.1 Procedure di riduzione

L'algoritmo di riduzione più semplice, chiamato *consistenza dei nodi*, riguarda i vincoli unari. Sia $r(X)$ un vincolo unario definito su una variabile X, e sia $rel(r)$ l'insieme dei suoi valori ammissibili. Senza perdita di generalità, il dominio di X può essere ristretto ai valori che appaiono in $rel(r)$; altrimenti, r è banalmente soddisfatto se $dom(X) \subseteq rel(r)$.
Consistenza dei nodi è basato sulla definizione di *2-consistenza di un valore*.

Definizione 31 (2-consistenza di un valore) *Sia X una variabile vincolata e sia $a \in dom(X)$ un valore. a viene detto 2-consistente (o consistente se non c'è rischio di ambiguità nel contesto) se per ogni variabile Y per cui esiste un vincolo $r(X, Y)$, esiste un valore $\beta \in dom(Y)$ tale che $r(a, b) \in rel(r)$. b viene chiamato supporto di a rispetto ad r.*

Esempio Il valore a è consistente per X (supportato da b per Y ed e per Z); il valore c non è consistente:

1. $p(X, Y)$ $rel(p) = \{\langle a, b\rangle, \langle c, d\rangle\}$;
2. $q(X, Z)$ $rel(q) = \{\langle a, e\rangle, \langle f, g\rangle\}$;

Chiaramente, se a non ha alcun valore di supporto rispetto a un qualche vincolo r che include X, allora X non può essere abbinata ad a. Perciò, tutti i valori non 2-consistenti possono essere rimossi da $dom(X)$. Questo risultato è stato ottenuto usando la *consistenza degli archi*; questo algoritmo considera ogni valore nel dominio di ogni variabile e rimuove tutti i valori non 2-consistenti.
La consistenza dei nodi e degli archi può essere generalizzata usando la k-consistenza:

Definizione 32 (k-consistenza di tuple di valori) *Una assegnazione parziale θ che assegna un valore a un sottoinsieme χ_θ delle variabili vincolate χ è consistente se non viola alcun vincolo, ossia soddisfa tutti i vincoli definiti su (un sottoinsieme di) χ_θ.*
Un CSP è k-consistente sse per qualsiasi assegnazione consistente θ su $k-1$ variabili, per qualsiasi variabile X non appartenente ad χ_θ esiste un valore a_X tale che $\theta' = \theta \cup \{X/a_X\}$ è consistente.

Se una soluzione parziale consistente di un insieme di $k-1$ variabili non può essere esteso ad un'altra variabile, allora questa soluzione parziale può essere rimossa (con tutte le sue specializzazioni) dallo spazio di ricerca delle assegnazioni. Questo algoritmo di potatura viene chiamato k-consistenza.

Casi particolari di k-consistenza sono i seguenti:

- 1-consistenza = consistenza dei nodi;
- 2-consistenza = consistenza degli archi;
- 3-consistenza = consistenza dei percorsi;

Per motivi di efficienza ed implementativi, la consistenza degli archi è generalmente preferita alla k-consistenza (con $k > 2$): da un lato, verificare la k-consistenza ha una complessità esponenziale in k; dall'altro, la k-consistenza mira a potare $(k-1)$-tuple di valori, il che richiede strutture dati specifiche per $k > 2$, la cui gestione potrebbe essere costosa dal punto di vista computazionale.

Complessità

Il miglior caso di complessità nel caso pessimo per un algoritmo di consistenza degli archi è $O(mL^2)$, con m numero dei vincoli e L dimensione del dominio delle variabili.

3.3.2 Procedure di ricerca

Gli algoritmi per risolvere CSP costruiscono, come soluzione, una assegnazione $\{X_i/a_i\}$ attraverso una visita in profondità dello spazio delle assegnazioni, a cui ci si riferisce come un albero delle sostituzioni. Ogni nodo corrisponde a una variabile X_i; l'arco di cui un estremo è questo nodo corrisponde a un valore candidato a_i che si potrebbe tentare di assegnare ad X_i. Per ogni assegnazione, viene verificata la consistenza; in caso di fallimento, viene considerato un altro valore candidato per il nodo corrente; se non sono disponibili altri valori, si procede al backtracking.

3.4 FASTheta

Definizione 33 (multisostituzione) *Un multilegame è denotato da $X \rightarrow T$, dove X è una variabile e $T \neq \emptyset$ è un insieme di costanti. Una multisostituzione è un insieme di multilegami $\Theta = \{X_1 \rightarrow T_1, \cdots, X_n \rightarrow T_n\} \neq \emptyset$, dove $\forall i \neq j : X_i \neq X_j$.*

Informalmente, un *multilegame* identifica un insieme di costanti che possono essere associate ad una variabile, mentre una *multisostituzione* rappresenta, in maniera compatta, un insieme di possibili sostituzioni per una tupla di variabili. In particolare, una singola sostituzione è rappresentata da una multisostituzione in cui ogni insieme di costanti è un singoletto (insieme contenente un unico elemento).

Esempio $\Theta = \{X \rightarrow \{1,3,4\}, Y \rightarrow \{7\}, Z \rightarrow \{2,9\}\}$ è una multisostituzione contenente 3 multilegami, ossia $X \rightarrow \{1,3,4\}$, $Y \rightarrow \{7\}$ e $Z \rightarrow \{2,9\}$.

Data una multisostituzione, l'insieme di tutte le sostituzioni che rappresenta può essere ottenuto scegliendo, in tutti i modi possibili, una costante per ogni variabile tra quelle del multilegame corrispondente.

Esempio L'insieme di tutte le sostituzioni rappresentate dalla multisostituzione $\Theta = \{X \rightarrow \{1,3,4\}, Y \rightarrow \{7\}, Z \rightarrow \{2,9\}\}$ è il seguente: $\{\{X \rightarrow 1, Y \rightarrow 7, Z \rightarrow 2\}, \{X \rightarrow 1, Y \rightarrow 7, Z \rightarrow 9\}, \{X \rightarrow 3, Y \rightarrow 7, Z \rightarrow 2\}, \{X \rightarrow 3, Y \rightarrow 7, Z \rightarrow 9\}, \{X \rightarrow 4, Y \rightarrow 7, Z \rightarrow 2\}, \{X \rightarrow 4, Y \rightarrow 7, Z \rightarrow 9\}\}$.

Definizione 34 (unione di multisostituzioni) *L'unione di due multisostituzioni* $\Theta' = \{\overline{X} \to T', X_1 \to T_1, \cdots, X_n \to T_n\}$ *e* $\Theta'' = \{\overline{X} \to T'', X_1 \to T_1, \cdots, X_n \to T_n\}$ *è la multisostituzione così definita:* $\Theta' \sqcup \Theta'' = \{\overline{X} \to T' \cup T''\} \cup \{X_i \to T_i\}_{1 \leq i \leq n}$.

Le due multisostituzioni devono essere definite sullo stesso insieme di variabili e devono differire al più in un multilegame.

Esempio L'unione di due multisostituzioni $\Sigma = \{X \to \{1,3\}, Y \to \{7\}, Z \to \{2,9\}\}$ e $\Theta = \{X \to \{1,4\}, Y \to \{7\}, Z \to \{2,9\}\}$ è $\Sigma \sqcup \Theta = \{X \to \{1,3,4\}, Y \to \{7\}, Z \to \{2,9\}\}$ (gli unici multilegami differenti sono quelli corrispondenti alla variabile X).

Definizione 35 (merge(S)) *Dato un insieme S di sostituzioni sulle stesse variabili, merge(S) rappresenta l'insieme di multisostituzioni ottenuto in accorso al seguente algoritmo:*

Require: *S: insieme di sostituzioni (ciascuna rappresentata come una multisostituzione).*
  while $\exists u, v \in S$ *tali che* $u \neq v$ *e* $u \sqcup v = t$ ***do***
    $S = (S\ \{u, v\}) \cup \{t\}$;
  end while
  return S;

In questo modo possiamo *fondere* 3 sostituzioni, rappresentandole con una sola multisostituzione.

Esempio $merge(\{\{X \to 1, Y \to 2, Z \to 3\}, \{X \to 1, Y \to 2, Z \to 4\}, (X \to 1, Y \to 2, Z \to 5\}\}) = merge\{\{X \to \{1\}, Y \to \{2\}, Z \to \{3,4\}\}, \{X \to \{1\}, Y \to \{2\}, Z \to \{5\}\}\}) = \{\{X \to \{1\}, Y \to \{2\}, Z \to \{3,4,5\}\}\}$.

Definizione 36 (intersezione di multisostituzioni) *L'intersezione di due multisostituzioni* $\Sigma = \{X_1 \to S_1, \cdots, X_n \to S_n, Y_1 \to S_{n+1}, \cdots, Y_m \to S_{n+m}\}$ *e* $\Theta = \{X_1 \to T_1, \cdots, X_n \to T_n, Z_1 \to T_{n+1}, \cdots, Z_l \to T_{n+l}\}$, *con* $n, m, l \geq 0$ *e* $\forall j, k : Y_j \neq Z_k$, *è la multisostituzione definita come:* $\Sigma \sqcap \Theta = \{X_i \to S_i \cap T_i\}_{i=1,\cdots,n} \cup \{Y_j \to S_{n+j}\}_{j=1,\cdots,m} \cup \{Z_k \to T_{n+k}\}_{k=1,\cdots,l}$ *sse* $\forall i = 1, \cdots, n : S_i \cap T_i \neq \emptyset$, *altrimenti non è definita.*

Esempio L'intersezione di $\Sigma = \{X \to \{1,3,4\}, Z \to \{2,8,9\}\}$ e $\Theta = \{Y \to \{7\}, Z \to \{1,2,9\}\}$ è $\Sigma \sqcap \Theta = \{X \to \{1,3,4\}, Y \to \{7\}, Z \to \{2,9\}\}$.

Esempio L'intersezione di $\Sigma = \{X \to \{1,3,4\}, Z \to \{8,9\}\}$ e $\Theta = \{Y \to \{7\}, Z \to \{1,2\}\}$ non è definita.

L'operatore $\sqcap$ è in grado di verificare se due multisostituzioni sono compatibili (ossia se condividono almeno una delle sostituzioni che rappresentano). In realtà, date due multisostituzioni Σ e Θ, se $\Sigma \sqcap \Theta$ non è definita, allora esiste almeno una variabile X comune a Σ e Θ a cui i multilegami corrispondenti associano insiemi di costanti disgiunti, il che significa che non esiste una costante che viene associata ad X sia da Σ che da Θ, e di conseguenza non può esistere una sostituzione comune. L'operatore $\sqcap$ può essere esteso ad insiemi di multisostituzioni: nello specifico, dati due insiemi di multisostituzioni S e T, la loro intersezione è definita come l'insieme di multisostituzioni ottenuto come segue:
$S \sqcap T = \{\Sigma \sqcap \Theta \mid \Sigma \in S, \Theta \in T\}$;
Da notare che, mentre una multisostituzione (e quindi l'intersezione di più multisostituzioni) è o non è definita, ma non può essere vuota, un insieme di multisostituzioni può esserlo. Quindi, una intersezione di insiemi di multisostituzioni può essere vuoto (ciò accade quando tutti gli insiemi che ne compongono l'intersezione non sono definiti).

Proposizione 3 *Sia* $C = \{l_1, \cdots, l_n\}$ *e* $\forall i = 1, \cdots, n : T_i = merge(uni(C, l_i, D))$*; sia* $S_1 = T_1$ *e* $\forall i = 2, \cdots, n : S_i = S_{i-1} \sqcap T_i$*.* C *θ-sussume* D *sse* $S_n \neq \emptyset$*.*

Questo risultato porta alla seguente procedura di θ-sussunzione:

Require: $C : c_0 \leftarrow c_1, c_2, \cdots, c_n, D : d_0 \leftarrow d_1, d_2, \cdots, d_m$ clausole.
  if $\exists \theta_0$ sostituzione tale che $c_0\theta_0 = d_0$ **then**
    $S_0 = \{\theta_0\}$;
    for $i = 1$ to n **do**
      $S_i = S_{i-1} \cap merge(uni(C, c_i, D))$;
    end for
  end if
  return $(S_n \neq \emptyset)$;

Esempio Date le seguenti sostituzioni: $\theta = \{X \to 1, Y \to 2, Z \to 3\}$, $\delta = \{X \to 1, Y \to 2, Z \to 4\}$, $\sigma = \{X \to 1, Y \to 2, Z \to 5\}, \tau = \{X \to 1, Y \to 5, Z \to 3\}$, una possibile sequenza di fusione è $(\theta \sqcup \delta) \sqcup \sigma$, che evita una ulteriore fusione con τ e porta al seguente insieme di multisostituzioni: $\{\{X \to \{1\}, Y \to \{2\}, Z \to \{3,4,5\}\}, \{X \to \{1\}, Y \to \{5\}, Z \to \{3\}\}\}$. Un'altra possibilità è fondere prima $\theta \sqcup \tau$ ed in seguito $\delta \sqcup \sigma$, che non possono essere fuse ulteriormente e portano a: $\{\{X \to \{1\}, Y \to \{2,5\}, Z \to \{3\}\}, \{X \to \{1\}, Y \to \{2\}, Z \to \{4,5\}\}\}$.

Questo algoritmo permette di effettuare una risoluzione tra clausole di Horn evitando il backtracking.

Require: un goal g, una teoria T ed una osservazione O
  if (g è unificabile con dei letterali in O) **then**
    $S \rightarrow merge(uni(C, g, O))$;
  else
    $S \rightarrow \emptyset$
    for all (clausola $C \in T$ tale che $testa(C)$ e g sono unificabili) **do**
      $C \rightarrow C\theta$ tale che $head(C)\theta = g$;
      $S' \rightarrow merge(\theta)$ /* viene applicata solo alle costanti in θ */
      for all ($l \in body(C)$) **do**
        $S' \rightarrow S' \sqcap Resolution(l, T, O)$;
      end for
    end for
    $S \rightarrow S \sqcup S'$
  end if
  $return(S)$;

Capitolo 4

Il Problema della Soddisfacibilità Booleana

4.1 Introduzione

Il problema della Soddisfacibilità Booleana (SAT) è un problema di decisione considerato nella teoria della complessità. Una istanza del problema è una espressione Booleana scritta usando solo le operazioni logiche $\wedge$, $\vee$ e $\neg$, variabili e parentesi. Il problema è: data l'espressione, esiste una assegnazione di valori $VERO$ e $FALSO$ alle variabili che la rendono vera ? Formalmente:

Definizione 37 (Problema della Soddisfacibilità Booleana) *Data una formula Booleana ϕ in Forma Normale Congiuntiva e contenente le variabili $(v_1,\ v_2,\ \cdots,\ v_n)$, esiste una assegnazione di valori $(c_1,\ c_2,\ \cdots,\ c_n)$, con $c_i \in \{VERO, FALSO\}$, $i = 1, \cdots, n$, tale che $\phi(c_1,\ c_2,\ \cdots,\ c_n) = VERO$?*

Esempio Sia $\phi(X_1, X_2, X_3)$ la formula Booleana in Forma Normale Congiuntiva $(X_1 \vee \neg X_2 \vee X_3) \wedge (X_2 \vee \neg X_1)$; una possibile assegnazione di valori di verità alle variabili della formula che la rendono vera potrebbe essere $\{X_1 = FALSO, = X_2 = FALSO,\ X_3 = VERO\}$, dato che $\phi(FALSO, FALSO, VERO) = (FALSO \vee \neg\ FALSO \vee VERO) \wedge (FALSO \vee \neg\ FALSO) \Leftrightarrow VERO$.

L'approccio del codificare in istanze di SAT le istanze di un dato problema è motivato dalla grande quantità di ricerca dedicata a questo problema negli ultimi anni e dal numero di *SAT solvers* disponibili: questi, sia completi che incompleti, sono in grado di risolvere istanze di centinaia di migliaia di clausole in pochi secondi, risultato inconcepibile fino a pochi anni fa. Inoltre, la comunità che lavora su SAT è ancora molto attiva, e i SAT solvers migliorano di giorno in giorno.

Teorema 2 (Cook-Levin) *Il problema della Soddisfacibilità Booleana è NP-completo, ossia:*

1. $SAT \in NP$;
2. SAT *è NP-hard;*

SAT è un problema NP-completo tipico, ed ogni istanza π di un problema in NP può essere tradotta in una istanza di SAT di grandezza polinomiale nella grandezza di π.

4.2 Algoritmi per SAT

A causa dell'importanza pratica della Soddisfacibilità Booleana, molto è stato fatto per sviluppare procedure di risoluzione efficienti per SAT per scopi pratici. In questa sezione, verrà fornita una breve descrizione di alcuni degli algoritmi più usati per risolvere SAT in maniera efficente.
Davis e Putnam hanno proposto un primo algoritmo basato su risoluzione nel 1960. L'algoritmo originale però soffriva di una elevata complessità in spazio: per risolvere questo problema, Davis, Logemann e Lovelan ne hanno proposta una variante nel 1962 che usava la ricerca al posto della risoluzione per limitarne il consumo di memoria.

4.2.1 Algoritmo Davis-Putnam

L'algoritmo per risolvere SAT proposto originariamente da Davis e Putnam è basato sulla risoluzione:

Risoluzione e Consenso

Sia S un insieme di letterali: $\sum S$ verrà usato per rappresentare la disgiunzione di tutti i letterali in S e $\prod S$ verrà usato per rappresentare la congiunzione di tutti i letterali in S. La legge del consenso stabilisce che è impossibile generare una clausola ridondante (cubo) da due clausole (cubi) se alcune condizioni sono soddisfatte.

Definizione 38 (Legge del Consenso) *Se S_1 e S_2 sono due insiemi di letterali, ed x è un letterale, allora:*

1. $x + \sum S_1)(x' + \sum S_2) \leftrightarrow (x + \sum S_1)(x' + \sum S_2)(\sum S_1 + \sum S_2)$;
2. $(x \prod S_1) + (x' \prod S_2) \leftrightarrow (x \prod S_1) + (x' \prod S_2) + (\prod S_1 \prod S_2)$;

Esempio Se a, b e c sono letterali, allora:

1. $(a+b)(a'+c) \leftrightarrow (a+b)(a'+c)(b+c)$;

2. $ab + a'c \leftrightarrow ab + a'c + bc$;

L'operazione di generare una clausola $(\sum S_1 + \sum S_2)$ dalle clausole $(x + \sum S_1)$ e $(x' + \sum S_2)$ è chiamata *risoluzione*. La clausola risultante $(\sum S_1 + \sum S_2)$ è chiamata *risolvente* delle clausole $(x+\sum S_1)$ e $(x'+\sum S_2)$. Nel nostro esempio, la clausola $(b+c)$ è il risolvente delle clausole $(a+b)$ e $(a'+c)$. L'operazione di generare il cubo $\prod S_1 \prod S_2$ dai cubi $x \prod S_1$ e $x' \prod S_2$ è chiamata *consenso*. Il teorema enunciato poco fa mostra che il risolvente di due clausole è ridondante rispetto alle clausole originali, così come il consenso di due cubi rispetto ai cubi originali.
La *distanza* tra due insiemi di letterali S_1 e S_2 è il numero di letterali l tali che $l \in S_1$, $l' \in S_2$. Nel teorema precedente, se la distanza tra S_1 e S_2 è maggiore di 1, allora il consenso di due cubi $x \prod S_1$ e $x' \prod S_2$ è un cubo vuoto, e il risolvente delle clausole $(x + \sum S_1)$ e $(x' + \sum S_2)$ è una tautologia. A causa della ridondanza delle tautologie e dei cubi vuoti, se non indicato altrimenti, risoluzione e consenso sono in genere limitati a due clausole/cubi con distanza 1. In questo caso, la distanza tra due clausole C_1 e C_2 è definita coma la distanza tra gli insiemi di letterali S_1 e S_2 dove $S_1(S_2)$ è l'insieme dei letterali che appare nella clausola $C_1(C_2)$. La distanza tra due cubi è definita in modo simile.

Due clausole possono essere risolte per generare una nuova clausola se la loro *distanza* è 1. La clausola risultante (chiamata clausola *risolvente*) contiene tutti i letterali che appaiono nelle due clausole originali eccetto il letterale che appare in entrambe con fasi differenti.
Date una formula in CNF ed una variabile x, le clausole nella formula possono essere divise in 3 insiemi:

1. A: clausole che non contengono x;

2. B: clausole che contengono x in fase positiva;

3. C: clausole che contengono x in fase negativa;

Date due clausole, una dall'insieme B ed una dall'insieme C, può essere generata una clausola risolvente priva della variabile x. Davis e Putnam hanno mostrato che se sono generate tutte le possibili clausole risolventi tra gli insiemi di B e C, vengono rimosse le tautologie tra di esse e le rimanenti

vengono congiunte con le clausole nell'insieme A, allora la formula CNF risultante avrà la stessa soddisfacibilità della formula originale con una variabile in meno.
Oltre alla risoluzione, l'algoritmo DP introduce una regola chiamata *regola del letterale unitario* per l'eliminazione di variabili. La regola afferma che se esiste una clausola nella formula CNF che contiene un solo letterale, allora la formula risultante dalla sostituzione di quel letterale con il valore $TRUE$ ha la stessa soddisfacibilità della formula originale.
Davis e Putnam hanno introdotto anche un'altra regola per l'eliminazione di variabili chiamata *regola del letterale puro*: una variabile (o letterale) viene definita *pura* se occorre solo in una singola fase nella formula CNF. La regola afferma che data una formula Booleana in CNF, la formula risultante dal cancellare tutte le clausole che contengono variabili (o letterali) pure ha la stessa soddisfacibilità della formula originale.
L'algoritmo DP applica queste due regole fin quando la formula non può essere ulteriormente semplificata. Quindi viene applicato il processo di risoluzione per eliminare una variabile dalla formula e di nuovo prova ad applicare le due regole alla formula risultante. Questo processo viene eseguito in maniera iterativa su una formula CNF ed elimina le variabili una ad una. Alla fine di questo processo iterativo, la formula non conterrà variabili. Se la formula risultante contiene una clausola vuota (clausola che non contiene letterali) allora la formula originale non è soddisfacibile. Se la formula risultante è una formula vuota (formula che non contiene clausole), allora la formula originale è soddisfacibile.
Nell'algoritmo DP, ogni volta che una variabile viene rimossa dalla risoluzione, nel caso pessimo il numero di clausole nella formula può crescere quadraticamente. Quindi, la complessità in spazio dell'algoritmo DP nel caso pessimo è esponenziale.

4.2.2 Algoritmo Davis-Logemann-Loveland

L'algoritmo di ricerca proposto da Davis, Logemann e Loveland [DLL62] è quello studiato più approfonditamente tra gli algoritmi per risolvere SAT. A differenza di DP e di altri algoritmi per risolvere SAT, la complessità in spazio di DLL è, di solito, predicibile. Grazie a questo, i risolutori basati su DLL possono gestire una grande quantità di formule senza un sovraccarico di memoria, e l'unico limite diventa il tempo richiesto per calcolare una soluzione. Questa è una proprietà desiderabile, dato che la maggior parte delle istanze di SAT sono di grandi dimensioni ma relativamente semplici da risolvere.
Lo spazio di ricerca di questo algoritmo viene spesso presentato come un al-

bero binario; all'interno dell'albero, ogni nodo rappresenta una assegnazione di un valore di verità ad una variabile applicato alla formula Booleana. I nodi foglia rappresentano delle assegnazioni complete (cioè tutte le variabili sono state assegnate) mentre i nodi interni rappresentano delle assegnazioni parziali (cioè alcune variabili sono state assegnate, le altre sono libere). Se alcuni nodi foglia vengono valutati come veri, allora la formula è soddisfacibile. La procedura di ricerca di DLL prova a cercare all'interno dell'intero albero e di determinare se tali nodi foglia esistono. Se viene trovato un nodo foglia valutato come vero, il risolutore produrrà il percorso dalla radice al nodo, che rappresenta l'assegnazione di variabili che soddisfa la formula; altrimenti, il risolutore riporterà che la formula è insoddisfacibile.
Ovviamente, sono necessarie alcune tecniche di potatura per evitare una enumerazione esponenziale di tutte le foglie nella ricerca di una soluzione. Per quasi tutti gli algoritmi basati su DLL, è richiesto che la formula in input sia in CNF. Una formula in CNF è soddisfatta se e solo se ognuna delle sue clausole viene soddisfatta individualmente. Una clausola è soddisfatta se e solo se almeno uno dei suoi letterali viene valutato come 1.
Da questo, derivano due importanti regole di ricerca:

1. **Regola del Letterale Unitario:** se una clausola non soddisfatta ha tutti i suoi letterali valutati come 0 tranne uno, allora il letterale libero restante deve essere valutato come 1 per rendere soddisfacibile la clausola (e l'intera formula CNF). Queste clausole vengono chiamate *clausole unitarie* e il letterale libero viene chiamato *letterale unitario*.

2. **Regola Conflittuale:** se una assegnazione parziale fa valutare 0 tutti i letterali di una clausola, allora non esiste alcuna foglia che soddisfa la formula all'interno del sotto-albero corrispondente a questa assegnazione parziale, ed il risolutore ha bisogno di tornare indietro ed esplorare altri rami in modo da trovare foglie che soddisfino la formula. Una clausola con tutti i suoi letterali valutati come 0 è chiamata *clausola conflittuale*.

Quando viene forzata una assegnazione da una clausola unitaria ad un letterale unitario a causa della regola del letterale unitario, il letterale si definisce *implicato*: l'attività che consiste nell'implicare letterali unitari viene chiamata *implicazione unitaria*. La clausola unitaria viene chiamata *antecedente* della variabile corrispondente al letterale implicato. Il processo iterativo che consiste nell'assegnare a tutti i letterali unitari il valore 1 fino a quando non ci sono più clausole unitarie viene chiamato *Propagazione dei Vincoli Booleani* (BCP, da Boolean Constraint Propagation).
BCP e backtracking sono le caratteristiche principali di un risolutore basato

su DLL: queste due semplici regole, insieme ad alcune altre tecniche, rendono DLL una procedura molto efficiente. La maggior parte (se non tutti) dei risolutori allo stato dell'arte sono basati su DLL. In letteratura, l'algoritmo DLL viene spesso presentato come una procedura ricorsiva. Lo pseudo-codice è il seguente:

Require: $formula, assignment$;
  $necessary = deduction(formula, assignment)$;
  $newAsgnmnt = union(necessary, assignment)$;
  if $isSatisfied(formula, newAsgnmnt)$ **then**
    $return(SATISFIABILE)$;
  else
    if $isConflicting(formula, newAsgnmnt)$ **then**
      $return(CONFLICT)$;
    end if
  end if
  $branchingVar = choseFreeVariable(formula, newasgnmnt)$;
  $asgn1 = union(newAsgnmnt, assign(branchingVar, 1))$;
  $result1 = DLL(formula, asgn1)$;
  if $result1 == SATISFIABLE$ **then**
    $return(SATISFIABLE)$;
  end if
  $asgn2 = union(newAsgnmnt, assign(branchingVar, 0))$;
  $return(DLL(formula, asgn2))$;

La funzione $DLL()$ viene invocata con una formula in CNF ed un insieme di assegnazioni di variabili come parametri. La funzione $detection()$ restituirà un insieme di assegnazioni di variabili necessarie che può essere dedotto dall'assegnazione di variabili esistente dalla regola del letterale unitario. Queste assegnazioni vengono aggiunte all'assegnazione di variabili originale creando un nuovo insieme di assegnazioni su cui verrà valutata la formula.
Se la formula è *soddisfatta* (cioè viene valutata come 1 o *vero*) o *conflittuale* (cioè viene valutata come 0 o *falso*) con l'assegnazione di variabili corrente, la ricorsione termina e viene restituito un risultato. Altrimenti, l'algoritmo sceglie una variabile non assegnata dalla formula, le assegna un valore vero o falso, aggiunge l'assegnazione all'insieme di assegnazioni corrente e invoca $DLL()$. Se il procedimento fallisce, il risolutore riprova assegnando un valore diverso alla variabile. Se entrambi i procedimenti sono falliti, la formula è insoddisfacibile. Il processo inizia invocando $DLL()$ con un insieme vuoto di assegnazioni di variabili.

4.2.3 Algoritmi Stocastici

Tutti i metodi e gli algoritmi per risolvere SAT descritti nella sezione precedente sono *completi*: dato un sufficiente tempo di esecuzione (e memoria), un algoritmo SAT completo può in ogni caso trovare una soluzione per una istanza di SAT o dimostrare che tale soluzione non esiste. Gli algoritmi SAT *stocastici* sono una classe differente di algoritmi SAT basati su tecniche di ottimizzazione matematiche: i metodi stocastici non possono dimostrare che una istanza di SAT non è soddisfacibile ma, data una istanza soddisfacibile, un SAT solver basato su metodi stocastici può trovare una soluzione in un tempo di esecuzione non eccessivo. Alcune applicazioni (vedi tesi per citazioni) possono utilizzare metodi SAT stocastici quando le istanze SAT tendono ad essere soddisfacibili, e dimostrarne la insoddisfacibilità non è necessario. Per alcune classi di benchmark, i metodi stocastici possono avere prestazioni significativamente migliori degli algoritmi completi esistenti che sono basati su una ricerca sistematica (ad es. l'algoritmo Davis-Logemann-Loveland, o DLL). In questa sezione, verranno discussi brevemente alcuni metodi stocastici per SAT.

Il problema SAT può essere visto come un caso specifico dei problemi di ottimizzazione combinatoria:

Definizione 39 (problema di ottimizzazione) *Definiamo un problema di ottimizzazione P_0 per mezzo di:*

1. *un insieme di istanze I_{P_0};*
2. *un insieme di soluzioni S_{P_0};*
3. *una proprietà di ammissibilità rappresentata dalla relazione di ammissibilità $R \subseteq I_{P_0} \times S_{P_0}$;*
4. *una funzione di misura $\mu : I_{P_0} \times S_{P_0} \mapsto \mathbb{N}$ tale che per ogni $x \in I_{P_0}, y \in S_{P_0}$, $\mu(x, y)$ è definita sse $y \in Sol(x)$;*

Un algoritmo per un problema di ottimizzazione P_0 calcola una funzione parziale $\chi : I_{P_0} \mapsto S_{P_0}$ tale che $\chi(x)$ è definita sse esiste almeno una soluzione $y = \chi(x)$ per la quale $\langle x, y\rangle \in \mathbb{R}$, e inoltre per ogni $z \in Sol(x)$, $\mu(x, y) \leq \mu(x, z)$ se $c = MIN$ e $\mu(x, y) \geq \mu(x, z)$ se $c = MAX$;

L'idea fondamentale di ottimizzazione basata su metodi stocastici per risolvere SAT consiste nel considerare il numero di clausole non soddisfatte come la funzione di misura, e nel tentare di rendere minima tale misura

assegnando valori alle variabili. Diverse euristiche ed algoritmi sviluppati per problemi generici di ottimizzazione combinatoria possono essere applicati alla risoluzione di SAT. Negli anni, sono state sperimentate diverse tecniche di ottimizzazione per SAT, come simulated annealing, algoritmi genetici, reti neurali e ricerca locale: tra tutti questi metodi, sembra che gli algoritmi basati sulla ricerca locale abbiano le prestazioni migliori per quanto riguarda SAT, e sono i più presenti in letteratura.
La ricerca locale è basata sul concetto della gara in salita. Lo spazio di ricerca è lo spazio Booleano delle combinazioni di assegnazioni per tutte le variabili: data una assegnazione a una variabile, l'azione fondamentale di un algoritmo di ricerca locale è di provare a spostarsi dall'assegnazione in modo da ridurre la funzione di misura o di allontanarsi da un minimo locale: di solito questo viene fatto cambiando il valore di una variabile assegnata, quindi la distanza di Hamming di due assegnazioni prima e dopo questa azione è di solito 1. I vari algoritmi di ricerca locale differiscono principalmente nel come scegliere la tale variabile e come allontanarsi dal minimo locale. GSAT e WalkSAT sono due algoritmi di ricerca locale rappresentativi.

GSAT

GSAT è un algoritmo proposto da Selman, Levesque e Mitchell nel 1992 [SLM92] ed è, in pratica, un semplice algoritmo greedy per minimizzare la funzione di misura.

```
Require:
  for (i = 1 to MAXTRIES) do
    T = assegnazione generata a caso;
    for (j = 1 to MAXFLIPS) do
      if (non esistono clausole non soddisfatte) then
        return(T);
      else
        x = trova una variabile in T con punteggio massimo;
        if (punteggio ≤ 0) then
          break;
        else
          cambia l'assegnazione di x;
        end if
      end if
    end for
    return(nessuna assegnazione soddisfacente trovata);
  end for
```

L'algoritmo inizia scegliendo una assegnazione ad una variabile casuale. In ogni passo, se l'istanza non è soddisfatta con la assegnazione corrente, l'algoritmo sceglie una variabile con un punteggio massimo (se sono più di una, tra queste viene applicata una scelta casuale) e ne cambia il valore. Il punteggio di una variabile x è la differenza tra il corrente numero di clausole non soddisfatte ed il numero di clausole non soddisfatte se il valore di x viene cambiato. Se nessuna variabile ha un punteggio positivo o è stato raggiunto il limite di passi MAXFLIPS, allora l'algoritmo si allontana da questo minimo locale ricominciando su una assegnazione ad una variabile casuale, fin quando non viene raggiunto il limite MAXTRIES.

WalkSAT

WalkSAT è stato proposto da McAllester, Selman e Kautz nel 1997 [MSK97]. La differenza principale tra WalkSAT e GSAT è nella selezione delle variabili di cui cambiare il valore nell'assegnazione. In WalkSAT, il punteggio di una variabile è il numero di clausole che cambieranno il proprio stato da soddisfatte a non soddisfatte se la variabile viene modificata. Se l'istanza non è soddisfatta, l'algoritmo sceglierà prima una clausola tra quelle non soddisfatte: se nella clausola selezionata esiste una variabile con punteggio 0 (ossia, modificarla non renderà non soddisfatta alcuna clausola soddisfatta) allora la variabile verrà modificata; altrimenti, con una certa probabilità, verrà modificata una delle variabili nella clausola con punteggio più alto. Questo viene chiamato *random walk*. Nel caso rimanente, verrà scelta una variabile casuale dalla clausola e modificata.
Nella pratica, spesso WalkSAT fornisce risultati migliori di GSAT dato che il rumore introdotto nel random walk è spesso sufficiente per allontanarsi dal minimo locale (al contrario, GSAT deve riavviarsi).

Capitolo 5

Riduzione del Problema della Theta-Sussunzione a SAT

In questo capitolo verrà trattato un metodo, migliorato per raffinamenti successivi, per trasformare le istanze del problema della θ-sussunzione in istanze del problema della soddisfacibilità Booleana.
Per comodità, denoteremo l'insieme delle variabili di una clausola X come $vars(X)$ e l'insieme dei suoi termini come $terms(X)$.

5.1 Spazio di Ricerca

In questo metodo, date due clausole Datalog C e D di cui si vuole verificare che esista un θ tale che $C\theta \subseteq D$, ogni variabile presente nell'istanza di SAT rappresenta un possibile binding tra una variabile $v \in vars(C)$ ed un termine $t \in terms(D)$: il numero di possibili variabili all'interno dell'istanza di SAT sarà quindi pari a $|vars(C)| \cdot |terms(D)|$.

Stabilendo una enumerazione delle variabili e dei termini in C e D, la variabile $V_{i,j}$ in una istanza SAT rappresenta il bind tra l'i-esima variabile in C e il j-esimo termine in D, e può assumere un valore tra $VERO$ e $FALSO$, a seconda dell'esistenza o meno di tale legame.

Per comodità, si considererà $|vars(C)| = m$ e $|terms(D)| = n$.

5.2 Ridurre lo Spazio di Ricerca

Abbiamo stabilito che la variabile $V_{i,j}$ dell'istanza SAT rappresenta il bind tra l'i-esima variabile in C e il j-esimo termine in D: ora è necessario stabilire dei vincoli sui possibili valori delle variabili dell'istanza I in modo tale che I sia soddisfacibile sse C θ-sussume D.

5.2.1 Un solo Legame per ogni Variabile

Un primo vincolo che è necessario porre consiste nel poter associare, ad ogni variabile in C, uno ed un solo termine in D, ossia: $\forall i \exists ! j : V_{i,j} = TRUE$.

Primo Metodo

Un primo modo per ottenere questo vincolo è tramite una formula strutturata nel seguente modo:

$$((V_{1,1} \wedge \neg V_{1,2} \wedge \cdots \wedge \neg V_{1,n}) \vee \cdots \vee (\neg V_{1,1} \wedge \cdots \wedge V_{1,k} \wedge \cdots \wedge \neg V_{1,n}) \vee \cdots \vee$$
$$(\neg V_{1,1} \wedge \cdots \wedge \neg V_{1,n-1} \wedge V_{1,n})) \wedge \cdots \wedge ((V_{l,1} \wedge \neg V_{l,2} \wedge \cdots \wedge \neg V_{l,n}) \vee \cdots \vee$$
$$(\neg V_{l,1} \wedge \cdots \wedge V_{l,k} \wedge \cdots \wedge \neg V_{l,n}) \vee \cdots \vee (\neg V_{l,1} \wedge \cdots \wedge \neg V_{l,n-1} \wedge V_{l,n})) \wedge \cdots \wedge$$
$$((V_{m,1} \wedge \neg V_{m,2} \wedge \cdots \wedge \neg V_{m,n}) \vee \cdots \vee (\neg V_{m,1} \wedge \cdots \wedge V_{m,k} \wedge \cdots \wedge \neg V_{m,n}) \vee \cdots \vee$$
$$(\neg V_{m,1} \wedge \cdots \wedge \neg V_{m,n-1} \wedge V_{m,n}));$$

Lo scopo di questa formula è di far legare ogni variabile $v \in vars(C)$ con un ed un solo termine $t \in terms(D)$.

Esempio Siano $C = \{p(X,Y),\ q(Y,Z)\}$ e $D = \{p(a,b),\ p(b,c),\ q(a,c),\ q(c,b)\}$ due clausole Datalog: i possibili binding dalle variabili di C ai termini di D sono i seguenti:

$V_{1,1} = \{X \to a\}$, $V_{1,2} = \{X \to b\}$, $V_{1,3} = \{X \to c\}$, $V_{2,1} = \{Y \to a\}$, $V_{2,2} = \{Y \to b\}$, $V_{2,3} = \{Y \to c\}$, $V_{3,1} = \{Z \to a\}$, $V_{3,2} = \{Z \to b\}$, $V_{3,3} = \{Z \to c\}$.

La formula sarà la seguente:

$$((V_{1,1} \wedge \neg V_{1,2} \wedge \neg V_{1,3}) \vee (\neg V_{1,1} \wedge V_{1,2} \wedge \neg V_{1,3}) \vee (\neg V_{1,1} \wedge \neg V_{1,2} \wedge V_{1,3}))\wedge$$
$$(V_{2,1} \wedge \neg V_{2,2} \wedge \neg V_{2,3}) \vee (\neg V_{2,1} \wedge V_{2,2} \wedge \neg V_{2,3}) \vee (\neg V_{2,1} \wedge \neg V_{2,2} \wedge V_{2,3})\wedge$$
$$(V_{3,1} \wedge \neg V_{3,2} \wedge \neg V_{3,3}) \vee (\neg V_{3,1} \wedge V_{3,2} \wedge \neg V_{3,3}) \vee (\neg V_{3,1} \wedge \neg V_{3,2} \wedge V_{3,3}));$$

Un problema di questo metodo è dovuto alla formula: questa può raggiungere una lunghezza eccessiva (n^2m^2 apparizioni di variabili, che nel caso di una conversione della formula in CNF potrebbe aumentare in maniera

esponenziale) ed inoltre non è in Forma Normale Congiuntiva. Per risolvere entrambi i problemi, la formula viene sostituita con altre due formule, così strutturate:

Secondo Metodo

Questa formula è costituita dalla congiunzione delle formule appartenenti ai seguenti insiemi:

1. $\{\neg V_{i,k} \vee \neg V_{i,h} \mid i = 1, \cdots, m \wedge k = 1, \cdots, n \wedge k < h\}$;

2. $\{V_{i,1} \vee V_{i,2} \vee \cdots \vee V_{i,n} \mid i = 1, \cdots, m\}$;

La prima formula assicura che ad ogni variabile di C venga associato un solo termine di D, ossia: $\forall V_{i,j}, V_{i,k} : V_{i,j} = V_{i,k} = TRUE \Leftrightarrow j = k$.

La seconda formula assicura che ad ogni variabile di C si associ almeno un termine di D, ossia: $\forall i \exists j : V_{i,j} = TRUE$.

Esempio Siano $C = \{p(X,Y),\ q(Y,Z)\}$ e $D = \{p(b,b),\ p(a,b),\ p(a,c),\ q(a,b),\ q(c,b)\}$ due clausole Datalog; le due formule generate saranno le seguenti:

1. $((\neg V_{1,2} \vee \neg V_{1,1}) \wedge (\neg V_{1,2} \vee \neg V_{1,3}) \wedge$
   $(\neg V_{1,1} \vee \neg V_{1,3}) \wedge (\neg V_{2,2} \vee \neg V_{2,1}) \wedge$
   $(\neg V_{2,2} \vee \neg V_{2,3}) \wedge (\neg V_{2,1} \vee \neg V_{2,3}) \wedge$
   $(\neg V_{3,2} \vee \neg V_{3,1}) \wedge (\neg V_{3,2} \vee \neg V_{3,3}) \wedge$
   $(\neg V_{3,1} \vee \neg V_{3,3}))$;

2. $((V_{1,2} \vee V_{1,1} \vee V_{1,3}) \wedge$
   $(V_{2,2} \vee V_{2,1} \vee V_{2,3}) \wedge$
   $(V_{3,2} \vee V_{3,1} \vee V_{3,3}))$;

Dunque le due formule assicurano che ad ogni variabile di C si associ uno ed un solo termine di D. Inoltre, le due formule sono in Forma Normale Congiuntiva (ossia non c'è bisogno di una conversione in CNF per fare in modo che l'istanza sia acquisibile dalla maggioranza dei SAT solvers) e la lunghezza complessiva della formula risultante dalla congiunzione delle due formule risulta di lunghezza $nm((m-1)/2+1) \leq n^2m^2$.

Chiameremo queste due formule, rispettivamente, α e β.

5.2.2 Vincoli Strutturali

Questi vincoli dipendono dalla struttura di C e D e servono per stabilire se, per ogni letterale $l \in C$, esiste una sostituzione θ tale che $l\theta \in D$.

Un primo vincolo che è necessario porre consiste nel poter associare, ad ogni variabile in C, uno ed un solo termine in D, ossia: $\forall i \exists ! j : V_{i,j} = TRUE$.

Primo Metodo

Questo metodo consiste nell'aggiungere, per ogni letterale $l \in C$, un insieme di formule corrispondenti a tutte le sostituzioni θ tali che $l\theta \in D$; il risultato è una formula che può essere calcolata dal seguente algoritmo:

Require: due clausole Datalog C e D.
  $F = \square$;
  for all letterale $l_i \in C$ **do**
    $F_i = \square$;
    for all letterale $l_j \in D$ **do**
      if esiste μ_{ij} tale che $l_i \mu_{ij} = l_j$ **then**
        $F_{ij} = \square$;
        for all legame $V \in \mu_{ij}$ **do**
          $F_{ij} = F_{ij} \wedge V$;
        end for
        $F_i = F_i \vee (F_{ij})$;
      end if
    end for
    $F = F \wedge (F_i)$;
  end for

In tale formula sono codificati i vincoli strutturali di C e D.

Esempio Siano C e D le clausole Datalog date nel precedente esempio: la formula sarà la seguente:

$$((V_{1,1} \wedge V_{2,2}) \vee (V_{1,2} \wedge V_{2,3})) \wedge ((V_{2,1} \wedge V_{3,3}) \vee (V_{2,3} \wedge V_{3,2}));$$

Secondo Metodo

Il precedente metodo un problema: la formula non è in Forma Normale Congiuntiva e, di conseguenza, non può essere usata come input per la maggior parte dei SAT solvers, ed una conversione in CNF, in alcuni casi, potrebbe aumentarne la dimensione in modo esponenziale. Per risolvere questo problema, la formula viene sostituita con una equivalente, questa volta in Forma

Normale Congiuntiva, che corrisponde alla seguente:

per ogni letterale di C di arità k verranno aggiunte delle formule del tipo:
$\neg V_{t_1,s_1} \vee \neg V_{t_2,s_2} \vee \cdots \vee \neg V_{t_k,s_k}$
che esprimono tutti i binding che non possono essere associati al letterale. In particolare, se per un letterale esistono tutti i possibili bindings, non ci sarà nessuna formula (ossia nessun vincolo).

Chiameremo questa formula γ.

Esempio Siano C e D le clausole Datalog $\{p(X,Y),\ q(Y,Z)\}$ e $\{p(b,b)$, $p(a,b)$, $p(a,c)$, $q(a,b)$, $q(c,b)\}$. La formula generata sarà la seguente:

$((\neg V_{1,2} \vee \neg V_{2,1}) \wedge (\neg V_{1,2} \vee \neg V_{2,3}) \wedge (\neg V_{1,1} \vee \neg V_{2,1}) \wedge (\neg V_{1,3} \vee \neg V_{2,2}) \wedge$
$(\neg V_{1,3} \vee \neg V_{2,1}) \wedge (\neg V_{1,3} \vee \neg V_{2,3}) \wedge (\neg V_{2,2} \vee \neg V_{3,2}) \wedge (\neg V_{2,2} \vee \neg V_{3,1}) \wedge$
$(\neg V_{2,2} \vee \neg V_{3,3}) \wedge (\neg V_{2,1} \vee \neg V_{3,1}) \wedge (\neg V_{2,1} \vee \neg V_{3,3}) \wedge (\neg V_{2,3} \vee \neg V_{3,1}) \wedge$
$(\neg V_{2,3} \vee \neg V_{3,3}));$

5.3 Soluzione

Il metodo risultante, quindi, consiste nella concatenazione delle tre formule α, β, γ:
$\alpha \wedge \beta \wedge \gamma;$
di cui, rispettivamente:

- α verifica che ad ogni variabile in C venga associato al più un termine in D;
- β verifica che ad ogni variabile in C venga associato almeno un termine in D;
- γ verifica che i vincoli strutturali tra C e D siano rispettati, ossia che, per ogni letterale $l_i \in C$, siano escluse dallo spazio delle soluzioni tutte le sostituzioni θ tali che $l_i \notin D$.

Esempio Nel caso delle seguenti clausole Datalog:

- C: $p(X,Y) : -p(X,Z), q(Y,Z), q(Z,Z).$
- D: $p(a,b) : -p(a,b), p(b,c), p(a,c), q(c,a), q(c,b), q(b,c), q(b,b), q(c,c).$

verificare che C θ-sussuma D equivale a risolvere la seguente istanza del problema della Soddisfacibilità Booleana:

$$\alpha = \begin{cases} (V_{1,1} \vee V_{1,2}) \wedge (V_{1,1} \vee V_{1,3}) \wedge (V_{1,2} \vee V_{1,3}) \wedge \\ (V_{2,1} \vee V_{2,2}) \wedge (V_{2,1} \vee V_{2,3}) \wedge (V_{2,2} \vee V_{2,3}) \wedge \\ (V_{3,1} \vee V_{3,2}) \wedge (V_{3,1} \vee V_{3,3}) \wedge (V_{3,2} \vee V_{3,3}) \wedge \end{cases}$$

$$\beta = \begin{cases} (V_{1,1} \vee V_{1,2} \vee V_{1,3}) \wedge (V_{2,1} \vee V_{2,2} \vee V_{2,3}) \wedge \\ (V_{3,1} \vee V_{3,2} \vee V_{3,3}) \wedge \end{cases}$$

$$\gamma = \begin{cases} (V_{1,1} \vee V_{3,1}) \wedge (V_{1,2} \vee V_{3,1}) \wedge (V_{1,2} \vee V_{3,2}) \wedge \\ (V_{1,3} \vee V_{3,1}) \wedge (V_{1,3} \vee V_{3,2}) \wedge (V_{1,3} \vee V_{3,3}) \wedge \\ (V_{2,1} \vee V_{3,1}) \wedge (V_{2,1} \vee V_{3,2}) \wedge (V_{2,1} \vee V_{3,3}) \wedge \\ (V_{2,2} \vee V_{3,1}) \wedge (V_{3,1}); \end{cases}$$

In questo caso, l'istanza di SAT ha due sole possibili soluzioni (ossia esistono due possibili θ tali che $C\theta \leq_\theta D$, e sono rispettivamente:

- $\{X \rightarrow a, Y \rightarrow b, Z \rightarrow b\}$;
- $\{X \rightarrow a, Y \rightarrow b, Z \rightarrow c\}$;

Che corrispondono alle uniche due soluzioni dell'istanza di SAT:

- $\{V_{1,1} = TRUE, V_{1,2} = FALSE, V_{1,3} = FALSE,$
  $V_{2,1} = FALSE, V_{2,2} = TRUE, V_{2,3} = FALSE,$
  $V_{3,1} = FALSE, V_{3,2} = TRUE, V_{3,3} = FALSE\}$;
- $\{V_{1,1} = TRUE, V_{1,2} = FALSE, V_{1,3} = FALSE,$
  $V_{2,1} = FALSE, V_{2,2} = TRUE, V_{2,3} = FALSE,$
  $V_{3,1} = FALSE, V_{3,2} = FALSE, V_{3,3} = TRUE\}$;

5.4 Esperimenti

Il nuovo algoritmo è stato implementato in C++ ed è costituito da due componenti principali:

- **SATheta**, che trasforma le istanze del problema della Theta-Sussunzione tra clausole Datalog in istanze di SAT;
- **MiniSat 2.0**, un SAT solver completo, sviluppato alla Chalmers tekniska högskola (CTH) e vincitore in tutte le categorie industriali nel concorso SAT 2005 [BW05]. MiniSat fa uso dell'algoritmo Davis-Putnam-Logemann-Loveland [DLL62], effettua backtracking tramite l'analisi dei conflitti e registrazione delle clausole [MSS96] e propagazione dei vincoli Booleani (BCP) [MMZ+01].

Per verificare l'efficienza del metodo, si è fatto ricorso alla *Transizione di Fase* [GBS99], un problema particolarmente difficile creato per studiare la complessità dell'analizzare formule nella logica del prim'ordine in un dato universo con lo scopo di trovarne i modelli, se ce ne sono.

Nella transizione di fase, ogni clausola ϕ viene generata a partire da n variabili (appartenenti ad un insieme X) e m predicati binari (appartenenti ad un insieme P), costruendo prima il suo *scheletro* $\varphi_s = \alpha_1(x_1, x_2) \wedge \cdots \wedge \alpha_{n-1}(x_{n-1}, x_n)$ (ottenuto concatenando le n variabili attraverso gli $(n-1)$ predicati) e quindi aggiungendo a φ_s i rimanenti $(m-n+1)$ predicati, i cui argomenti vengono selezionati da X.
Dato Λ, un insieme di L costanti, viene generato un esempio usando N letterali per ogni simbolo di predicato in P, su cui verificare la sussunzione della clausola generata: gli argomenti dei letterali sono selezionati dall'universo $U = \Lambda \times \Lambda$. Questo problema di matching può essere definito dalla tupla (n, m, L, N).

Seguendo le linee guida in [MM01], è stato creato un insieme di coppie $(regola, esempio)$: come in [MM01], n è stato impostato a 10, m spazia in [10, 60] ed L spazia in [10, 50]. Per limitare il costo computazionale totale, N è stato impostato a 64 anzichè a 100: questo non influisce sulla presenza del fenomeno della transizione di fase, ma riduce il numero delle possibili soluzioni.

Per ogni coppia (m, L), sono state generate 33 coppie $(regola, esempio)$, ed è stato calcolato il costo computazionale medio, in secondi, della procedura di θ-sussunzione.

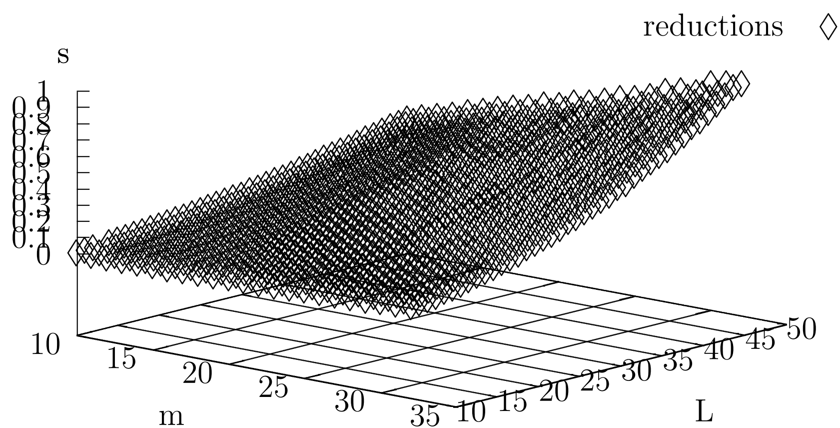

Figura 5.1: Riduzioni delle istanze del problema della θ-sussunzione in istanze di SAT: è possibile notare che il tempo richiesto dall'algoritmo di traduzione delle istanze cresce polinomialmente rispetto ad L (numero di termini presenti all'interno dell'esempio) ed m (simboli di predicato nella regola).

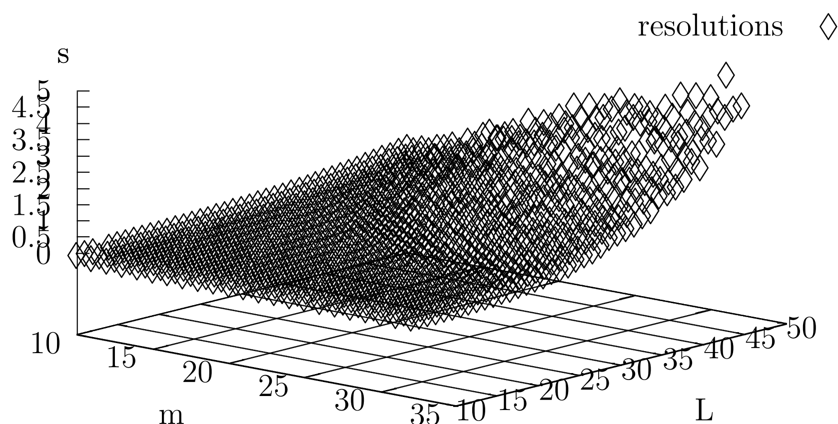

Figura 5.2: Risoluzioni delle istanze di SAT tramite MiniSat: è possibile notare che il grafico somiglia molto ad una superficie parabolica, ed il tempo richiesto per la risoluzione delle istanze cresce esponenzialmente rispetto ad L ed m: la causa di ciò dipende dal fatto che tutti gli algoritmi conosciuti per risolvere problemi *NP*-completi hanno complessità superpolinomiale. Per far fronte al problema, è possibile rinunciare alla completezza del SAT solver ed usare algoritmi per SAT stocastici (come GSAT [SLM92] o WalkSAT [MSK97])che, come si è visto, sono in grado di trovare una soluzione per una istanza, se questa esiste, in un tempo non eccessivo.

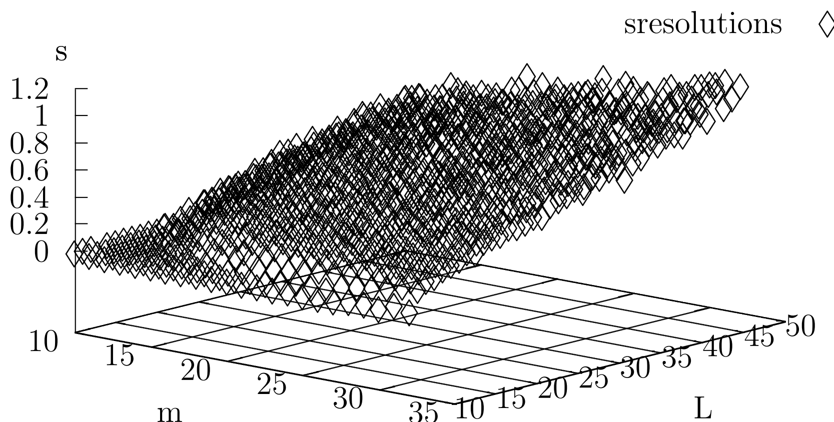

Figura 5.3: Risoluzioni delle istanze di SAT tramite GSAT [SLM92]; GSAT è un algoritmo stocastico per risolvere istanze del problema della Soddisfacibilità Booleana: parte assegnando valori random a tutte le variabili; se l'assegnazione soddisfa tutte le clausole, l'algoritmo si ferma e restituisce l'assegnazione. Alterimenti, il valore di una variabile viene cambiato, e il processo si ripete. La variabile che cambia è quella che minimizza il numero di clasuole non soddisfatte nella nuova assegnazione: se non viene trovata alcuna assegnazione in grado di soddisfare tutte le clausole dopo un numero prefissato di iterate, l'algoritmo riparte con una nuova assegnazione casuale. L'algoritmo termina o quando viene trovato un modello della formula o quando il numero di riavvii eccede un valore prefissato (in questo caso 0, ossia non vi sono stati riavvii). Dal grafico si può notare che l'algoritmo ha una complessità temporale molto inferiore rispetto ad uno completo, derivante dal fatto che, a volte, può non essere trovata una soluzione di una istanza di SAT soddisfacibile.

5.5 Conclusioni

Con questa tesi, si è voluto dimostrare che le riduzioni in tempo polinomiale (o Karp-riduzioni) possono essere usate efficacemente per trovare soluzioni efficienti a problemi in *NP*: in questo caso, partendo dalla definizione di un generico problema appartenente alla suddetta classe di complessità, è stato possibile, tramite l'analisi del suo spazio delle soluzioni, creare un algoritmo in grado di trasformarne le istanze in istanze in istanze di un classico proble-

ma *NP*-completo (in questo caso il Problema della Soddisfacibilità Booleana), per cui esistono già soluzioni efficienti (complete e non).

Bibliografia

[Bax76] L. D. Baxter. The complexity of unification. *Doctoral Thesis, Dept. of Computer Science, University of Waterloo, Waterloo, Ontario*, 1976.

[Bax77] L. D. Baxter. The np-completeness of subsumption. *unpublished manuscript*, 1977.

[BW05] Fahiem Bacchus and Toby Walsh. Eighth international conference on theory and applications of satisfiability testing. University of St. Andrews Conference Centre, St. Andrews, Scotland, 2005.

[DD97] L. Deraedt and L. Dehaspe. Clausal discovery. *Machine Learning*, 26:99–146, 1997.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[DMR92] Saso Dzeroski, Stephen Muggleton, and Stuart J. Russell. PAC-learnability of determinate logic programs. In *Proceedings of the Fifth Annual ACM Workshop on Computational Learning Theory (COLT-92)*, Pittsburgh, Pennsylvania, 1992. ACM Press.

[GBS99] Attilio Giordana, Marco Botta, and Lorenza Saitta. An experimental study of phase transitions in matching. In *IJCAI '99: Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 1198–1203, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[Got87] Georg Gottlob. Subsumption and implication. *Inf. Process. Lett.*, 24(2):109–111, 1987.

[KL94] J-U. Kietz and M. Lübbe. An efficient subsumption algorithm for inductive logic programming. In S. Wrobel, editor, *Proceedings of the 4th International Workshop on Inductive Logic Programming*, volume 237, pages 97–106. Gesellschaft für Mathematik und Datenverarbeitung MBH, 1994.

[KN86] Deepak Kapur and Paliath Narendran. Np-completeness of the set unification and matching problems. In *Proceedings of the 8th International Conference on Automated Deduction*, pages 489–495, London, UK, 1986. Springer-Verlag.

[MF90] S. Muggleton and C. Feng. Efficient induction of logic programs. In *Proceedings of the 1st Conference on Algorithmic Learning Theory*, pages 368–381. Ohmsma, Tokyo, Japan, 1990.

[MM01] J. Maloberti and Sebag M. Theta-subsumption in a constraint satisfaction perspective. In *Proceedings of Inductive Logic Programming*, pages 164–178. Springer Verlag LNAI 2157, 2001.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, 2001.

[MSK97] David McAllester, Bart Selman, and Henry Kautz. Evidence for invariants in local search. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97)*, pages 321–326, Providence, Rhode Island, 1997.

[MSS96] Joao P. Marques-Silva and Karem A. Sakallah. GRASP - A New Search Algorithm for Satisfiability. In *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pages 220–227, 1996.

[Mug92] S. Muggleton. Inverting implication. In S. Muggleton, editor, *Proceedings of the 2nd International Workshop on Inductive Logic Programming*, pages 19–39, 1992.

[Plo70] G.D. Plotkin. A note on inductive generalization. In *Machine Intelligence*, volume 5, pages 153–163. Edinburgh University Press, 1970.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.

[SHW96] Tobias Scheffer, Ralf Herbrich, and Fritz Wysotzki. Efficient theta-subsumption based on graph algorithms. In *Inductive Logic Programming Workshop*, pages 212–228, 1996.

[SLM92] Bart Selman, Hector J. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In Paul Rosenbloom and Peter Szolovits, editors, *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 440–446, Menlo Park, California, 1992. AAAI Press.

[SS88] Manfred Schmidt-Schauss. Implication of clauses is undecidable. *Theor. Comput. Sci.*, 59(3):287–296, 1988.

[vdLNC93] Patrick R. J. van der Laag and Shan-Hwei Nienhuys-Cheng. Subsumption and refinement in model inference. In *European Conference on Machine Learning*, pages 95–114, 1993.

[WKS81] F. Wysotzki, W. Kolbe, and J. Selbig. Concept learning by structured examples - an algebraic approach. In *Proc. of the 7th IJCAI*, pages 153–158, Vancouver, Canada, 1981.