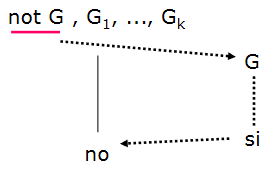
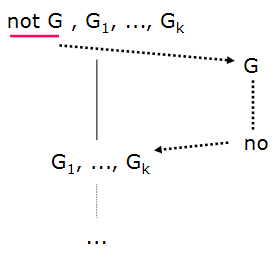
Riassunto di problem solving dichiarativo

# Negazione nella programmazione logica

La negazione potrebbe essere utile nelle formule che utilizzano il taglio per farle diventare più modulari, inoltre permettono di sapere quando una certa proprietà non vale.

Come si fa? si utilizza not seguito da un predicato, esso rovescia la risposta data da quest’ultimo, precisamente:

* Dato un predicato G, se ha successo, allora not G fallisce;
* se invece G fallisce, not G ha successo.

<Potrebbe essere scritto meglio>

Proprio per questo motivo, questo tipo di negazione è detta negazione per fallimento, che differenza c’è rispetto a quella logica?

## Prima differenza

Dato un insieme S di formule, se da esse non segue una formula G, non è detto che segue not G. Scartando i casi in cui S è inconsistente, ci sono tre possibilità per G:

* dare sì come risposta, quindi da S segue G;
* dare no come risposta, quindi da S segue not G;
* dare non so come risposta, in quest’ultimo caso non seguono nè G nè not G da S, la conoscenza è quindi incompleta.

### Esempio

Considerando la formula a or b, da essa non seguono logicamente nè a, nè b dato che queste non sono vere in ogni interpretazione e sono falsi in almeno un caso, l’informazione è quindi incompleta.

## Come si procede?

In alcuni casi è possibile ammettere un certo insieme di regole come conoscenza completa, come si fa? ebbene, non si ammette “non so” come risposta.

Risulta quindi più comodo scrivere in quali casi le relazioni sono vere e lasciare quelle false sottintese.

Considerando le regole p(X):- q(X) e p(X):-r(X):

* queste risulterebbero essere come p(X) ← q(X) or r(X);
* in realtà però si avvicina di più a p(X) ↔ q(X) or r(X);

Con quest’ultima infatti si riuscirebbe a concludere anche che p(X) è falso se q e r lo sono per X e viceversa.

## Seconda differenza

La seconda differenza riguarda le clausole: data la regola A:- L1,...,Ln in cui i vari Li possono essere positivi o con not, le conclusioni degli Li verranno utilizzate per trovare una conclusione di A, quindi:

* si conclude A se si conclude ogni Li;
* si conclude not A se almeno un Li fallisce, di conseguenza si conclude not (L1,...,Ln) e falliscono tutti gli altri modi per concludere A.

Considerando le regole precedenti, il ← implicito in p(X) → q(X) or r(X) viene utilizzato per concludere not p(a) da not q(a) and not r(a) ma non per concludere q(a) da p(a) e not r(a).

## Terza differenza

La terza differenza riguarda i goal, il significato di un goal not G è infatti diverso e ciò è dovuto al fatto che le variabili in G sono legate o meno a dei valori.

Considerando la regola q(X):- not p(X), ciò può essere inteso come not ∃p(X) se chiamato con X variabile, quindi:

* si chiama ∃Xp(X);
* si rovescia la risposta.

Normalmente un goal G in cui è presente X viene inteso come ∃X:G, la negazione viene quindi applicata solo a G e non anche a X.

### Esempio

Si considera l’insieme di regole a destra, da esse si può dire che:

* un disoccupato è un adulto che non è occupato;
* un occupato può essere uno studente oppure un dipendente di un’azienda;
* mario e giovanni sono adulti, quest’ultimo inoltre è dipendente dell’azienda acme.

Che considerazioni si possono fare?

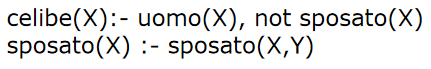
* disoccupato(mario) ha successo dato che verifica sia adulto(mario), sia not occupato(mario);
* disoccupato(X) però fallisce invece di dare X=mario, questo perchè not occupato(X) avrebbe successo nel caso in cui nessuno è occupato, quindi not ∃X:occupato(X).

A fronte di ciò, come si risolve questo problema? Si invertono le precondizioni nella prima regola, in questo modo si valuta prima se X è adulto e poi se non è occupato.

In questo modo, disoccupato(X) ha successo e trova X=mario più eventuali soluzioni analoghe.

In alcuni sistemi di programmazione logica, si può utilizzare una regola di selezione dinamica per ritardare i goal negativi con variabili non legate, in questo modo vengono ritardati anche gli errori runtime se non ci sono altri goal da scegliere.

### Esempio

Considerando ora le regola in alto a destra, un celibe X è un uomo che non è sposato con Y,

ciò funziona perchè uomo(X) fa in modo che X venga legato a un valore.

La seconda precondizione intende invece che non esiste un Y per cui X è sposato.

Il secondo insieme di regole dice essenzialmente la stessa cosa ma permette di evitare dubbi, la lettura infatti rimane equivalente alla quantificazione universale.

# Metaprogrammazione

Nella programmazione logica è possibile utilizzare i goal come argomenti, in questi casi non esiste una distinzione sintattica.

L’esempio più semplice è call(X), la sua esecuzione indica semplicemente la chiamata del goal X, da ciò è possibile definire not(X) come segue:

* not(X):- call(X),!,fail;
* not(X).

Se la chiamata ha successo, call(X) è positivo e fa il taglio, quest’ultimo è quello che fa fallire not(X) nel caso in cui call(x) fallisca.

L’introduzione di call è stata fatta per leggibilità, in alcuni sistemi come ECLiPSe si può anche omettere.

### Esempio

Si può creare un predicato che indichi che un goal abbia successo al massimo una volta, come si fa? Per fare ciò si procede come segue:

* once(X):- call(X),!.

CIò può risultare utile ad esempio per trovare solo la prima occorrenza di una lista senza trovare tutte le possibili soluzioni.



Questa risulta una soluzione migliore rispetto alla definizione di varianti di member, ciò permette di evitare tanti predicati che fanno essenzialmente la stessa cosa.

Esistono anche altre relazioni che permettono di costruire termini o goal o di prenderne i dati:

* functor(T,P,N) indica che il termine T ha P come simbolo principale e N argomenti, ad esempio:

functor(p(a,b),P,N) P=p N=2

* arg(N,T,X) indica che il termine T ha X come N-esimo argomento, ad esempio:

arg(1,p(a,b),X) X=a

# Eccezioni

Si può rappresentare la conoscenza attraverso delle eccezioni? Certo! Ma come si modellano? Si utilizza un predicato in cui si specificano i singoli casi.

Tutto ciò si può modellare anche senza l’utilizzo delle eccezioni, tuttavia verrebbe più complesso.

## Vincoli

I vincoli sono variabili o simboli di funzioni/relazioni che devono essere rispettati al fine di ottenere un risultato, in essi le variabili fungono da placeholder per i valori con cui sostituirle.

Questi vincoli possono essere:

* primitivi quando sono relazioni costruite con variabili e costanti, X>4 e X+2Y=9 ne sono degli esempi;
* normali quando sono congiunzioni di vincoli primitivi, come ad esempio X>4 & X+2Y=9.

La soddisfacibilità di un vincolo si basa su due concetti:

* valutazione, un assegnamento di valori alle variabili;
* soluzione, una valutazione che soddisfa i vincoli.

Da ciò si può dire che un vincolo è soddisfacibile se esiste una soluzione, cioè una valutazione che lo soddisfa, è invece insoddisfacibile se la soluzione non esiste.

Davanti a un vincolo bisogna porsi due domande:

* Esiste una soluzione?
* Se esiste, come si ricava?

La risposta alla prima domanda è semplice e consiste nell’utilizzo di un solver di vincoli.

Per quanto riguarda la seconda, si potrebbe andare per tentativi, ciò però è controproducente nei domini infiniti, in questi casi si utilizzano dei metodi di enumerazione, si lavora su domini finiti oppure su metodi in generale “non casuali”.

Attraverso la semplificazione, è possibile rappresentare un vincolo in maniera più semplice oppure trovare una soluzione a un sottoinsieme di variabili.

## Eliminazione di Gauss-Jordan

* Si sceglie un’equazione c da un insieme C;
* si riscrive c in modo equivalente nella forma x=e, dove e non contiene x;
* si continua così finchè tutte le equazione non sono nella forma x=e (restituendo true) oppure se si trova una formula equivalente a d=0, in tal caso si restituisce false perchè d deve essere diverso da 0.

Questo metodo permette di capire se esiste un modello e in tal caso restituire la soluzione.

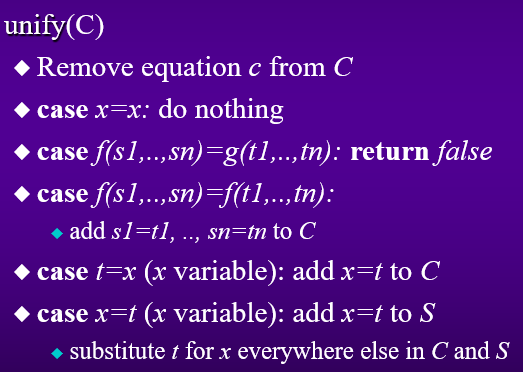
La soluzione restituita è nella cosiddetta forma risolta, quest’ultima è formata da:

* variabili non parametriche, cioè quelle che occorrono nella parte sinistra di un equazione;
* variabili parametriche, quelle che occorrono a destra in un numero arbitrario di equazioni;
* soluzioni: scelgono i valori dei parametrici e determinano i non parametrici.

## Vincoli ad albero

I vincoli ad albero rappresentano un dato strutturato e risultano utili nell’unificazione dei termini, risolvendo tutto in un modo simile a Gauss-Jordan:

* si parte da un set di termini di equazioni C e un set vuoto di termini S;
* si continua fino a quando C diventa vuoto oppure viene restituito false.

La funzione unify(C) a destra indica tutti i possibili casi:

* prima di tutto si rimuove l’equazione c dall’insieme C;
* se c è nella forma x=x, non si fa niente;
* se invece è nella forma f(s1,...,sn)=g(t1,...,tn), l’equazione non è risolvibile, si restituisce false;
* se invece la forma di c è f(s1,...,sn)=f(t1,...,tn), allora si aggiungono a C i vari si=ti;
* se c è t=x, dove x è una variabile, allora si aggiunge x=t a C;
* se c è x=t con x variabile, si aggiunge x=t a S, quel che si fa è sostituire le t con le x in ogni equazione di C e S.

## Cos’è un risolutore?

Un risolutore di vincoli è una funzione che prende un vincolo c e restituisce true, false oppure non so in base alla soddisfacibilità, infatti:

* viene restituito true se c è soddisfacibile;
* viene restituito falso se c è insoddisfacibile;
* viene restituito non so se il risolutore non sa se c è soddisfacibile o meno.

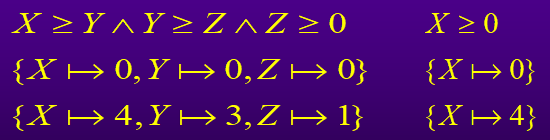
Un risolutore è completo se restituisce solamente true o false.

Il linguaggio accettato dal risolutore di vincoli può essere limitato al caso, per le equazioni di Gauss-Jordan ad esempio si è limitati ai casi lineari, tutti gli altri che con rispettano questo linguaggio verranno etichettati come “non so”.

Proprio per questo motivo, tutto ciò rende interessante il lavoro di questi solver.

## Proiezione

La proiezione di un vincolo C su un insieme V di variabili è un vincolo C1 tale che:

* contiene solo variabile in V;
* ogni soluzione di C è una soluzione di C1;
* una soluzione di C1 può essere estesa per diventare una soluzione di C.

Considerando le formule a sinistra nella foto, per V={X} si ottengono le proiezioni a destra, cioè tutti i casi in cui occorrono le variabili in V.

## Algoritmo di Fourier

L’algoritmo di Fourier permette di eliminare da una disequazione lineare C, come fa a farlo? Ebbene, l’algoritmo scrive le disequazioni nel seguente modo:

Per ogni e , si crea una terza formula combinando le prime due:

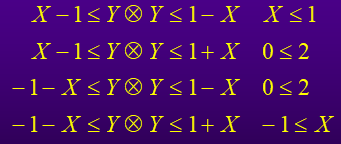
Come risultato si ottengono tutte le disequazioni vecchie e nuove in C che non involvono in y.

### Esempio

Considerando la foto a destra, si estraggono da essa le seguenti disequazioni:



Combinando insieme ogni possibile combinazione, si ottengono le seguenti disequazioni:



Da ciò, si possono quindi fare le seguenti considerazioni:

* L’insieme di disequazioni di partenza equivale al quadrato nella figura sopra;
* L’insieme ottenuto con l’algoritmo di Fourier invece equivale alla linea orizzontale nella figura, ciò indica che l’insieme ottenuto è un sottoinsieme di quello di partenza in cui si considera solamente la variabile X e nessun’altra.

## Cos’è un semplificatore?

Un semplificatore è una funzione che, dato un vincolo C, restituisce un vincolo C1 con lo stesso significato ma limitato a un insieme di variabili V.

Due vincoli C1 e C2 sono equivalenti per un insieme V se ogni soluzione di C1 ristretta a V può essere riestesa per C2 e viceversa.

Un semplificatore ha le seguenti proprietà desiderabili:

* proiezione: le variabili di un semplificatore devono essere un sottoinsieme di V;
* proiezione debole: per ogni vincolo C2 equivalente a C1 per V, il numero di variabili del semplificatore per C1 deve essere minore o uguale al numero di variabili in C2.

Ciò che si vuole ottenere è quindi l’utilizzo delle sole variabili che effettivamente servono all’interno dei vincoli,

# Regole

L’idea del constraint logic programming consiste nella definizione di regole le quali definiscono un modello.

Per fare un esempio, si considera il circuito a destra, da esso si ricavano le seguenti variabili:

* V indica la tensione della batteria;
* I, I1 e I2 indicano le correnti rispettivamente totale e sui due resistori;
* R1 e R2 sono le variabili indicanti i valori di resistenza.

L’emulazione del circuito avviene definendo una regola con tre vincoli:

* parallel\_resistors(V,I,R1,R2):-V=I1\*R1, V=I2\*R2, I1+I2=I

Queste condizioni devono valere affinchè i valori ottenuti abbiano un senso.

In queste regole ci saranno quindi vincoli user-defined e altri risolti dal risolutore.

## Vincoli user defined

I vincoli user defined sono vincoli definiti come un predicato avente n argomenti, ogni letterale di questo vincolo può essere un vincolo primitivo o anch’esso user defined.

Un goal è una sequenza di m letterali, una regola A:-B indica che A è un vincolo user-defined mentre B è il goal.

Si può quindi concludere che un programma è una sequenza di regole.

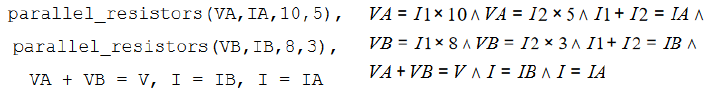
L’applicazione di una regole impone una ridenominazione delle variabili per evitare che vengano utilizzate le stesse.

La ridenominazione è un mapping biettivo da un insieme di variabili a un altro.

Un’oggetto sintattico può essere un vincolo user-defined o primitivo, un goal oppure una regola.

L’applicazione di una ridenominazione a un oggetto sintattico rimpiazza ogni variabile dell’oggetto con una nuova.

Una variante o’ di un oggetto è un particolare oggetto per cui r(o’)=o.



## Uso dei vincoli nella programmazione logica

Nella programmazione logica, i vincoli si indicano con un’espressione, ad esempio:

* T is X+Y+Z, T<10: T è la somma delle variabili X, Y e Z e deve essere minore di 10.

Si considera ora come esempio il fattoriale, come si rappresenta con la logica?

* fact(0,1).
* fact(N,N\*F):- {N>=1}, fact(N-1,F).

La prima regola è il caso base e si applica quando N=0 e F=1, la seconda invece è il caso più generale, in questo caso F=N-1 e ciò spiega perchè c’è N\*F nella premessa della regola.

La chiamata ricorsiva viene quindi fatta solo su F dal momento che N è già stato verificato.

Il vincolo N>=1 deve essere rispettato per fare in modo che tutto funzioni come deve.

### Valutazione

A ogni passo si interroga il risolutore per verificare se la congiunzione di vincoli è soddisfacibile, come si fa?

Si selezione un letterale a ogni passo:

* se il letterale è un vincolo user-defined, allora si riscrive;
* se invece è un vincolo primitivo, si aggiunge a una memoria di vincoli.

Questa valutazione è considerata come un’estensione della risoluzione SLD.

Detto ciò, si può considerare tutto nel seguente modo:

* a ogni step viene definito uno stato <G,C> in cui G è un insieme di goal e C un insieme di vincoli inizialmente vuoto;
* un passo di derivazione prende un letterale e da G e valuta se si può risolvere con gli attuali vincoli in C, se è così allora si continua, altrimenti il risolutore si ferma, non esiste una soluzione;
* Nel caso in cui il letterale scelto sia un vincolo user-defined, allora l’insieme G viene riscritto utilizzando regole e ridenominazioni, l’insieme C invece rimane tale.

La derivazione parte da uno stato <G,true> ed è una sequenza di passi di derivazione da uno stato all’altro.

Uno stato è di successo se è nella forma <[],C> dove C è soddisfacibile oppure non si sa, altrimenti è uno stato di fallimento.

Una derivazione è di successo se l’ultimo stato è di successo, lo stesso discorso vale per una derivazione di fallimento.

### Alberi di derivazione

Un albero di derivazione per un insieme di goal G è un albero avente <G,true> alla radice e i figli di ogni stato sono tutti gli stati raggiungibili con uno step di derivazione., esso indica tutte le possibili derivazioni.

In questo albero se il letterale leftmost è un vincolo primitivo, allora lo stato avrà un solo figlio, altrimenti ne avrà tanti ordinati in base alle regole.

### Esempio

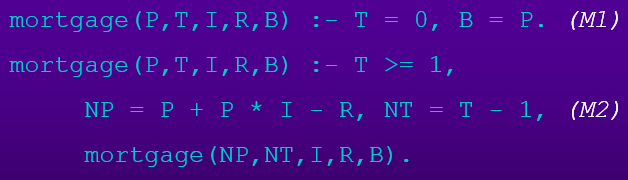
Dato un prestito P, un tasso di interesse I, una data periodica R e un bilancio B su T periodi, come si può modellare il predicato montagage?

* Nel caso dell’interesse semplice, B=P+P\*I-R;
* Con T periodi invece, si effettua lo stesso calcolo T volte;

Sulla base di ciò, come si definisce il predicato?

* montgage(P,0,I,R,B) :- {B=P}.
* montgage(P,T+1,I,R,B) :- {N=P+P\*I-R}, montgage(N,T,I,R,B).

Un’altra soluzione è la seguente:



## Ottimizzazione

Dato che alcuni problemi richiedono la miglior soluzione che minimizza una data espressione E, cosa si può fare?

Volendo si possono ridefinire i predicati integrando la minimizzazione, tuttavia risulta più conveniente utilizzare minimize(G,E), in essa:

* si trova una soluzione per il goal G;
* una volta trovata, si impone un vincolo in cui la valutazione di E per G deve essere minore della soluzione trovata in precedenza.

### Esempio

Dato i predicati leaf e node per indicare rispettivamente foglie e nodi, come si trovano le foglie di un albero e il loro livello?

* Caso base: leaf(node(null,X,null),X,0), questo predicato indica un albero con un solo nodo, la radice;
* Per quanto riguarda i casi ricorsivi, si vanno a vedere i due sottoalberi:

leaf(node(L,\_,\_),X,D+1):-leaf(L,X,D).

leaf(node(\_,\_,R),X,D+1):-leaf(R,X,D).

Le foglie dell’albero stanno per definizione a livello 0 e appartengono a un albero con un solo nodo, quelle a livello D invece sono in un sottoalbero L1 a livello D e, di conseguenza, appartengono a un albero L a livello D+1.

# Problem solving dichiarativo

Il problem solving dichiarativo indica la descrizione di un problema per poterne trovare una soluzione con un procedimento automatico, il tutto senza implementare un algoritmo ad-hoc.

Ciò è presente nella programmazione logica basata su SLD e vincoli numerici, qui infatti è presente l’invertibilità degli argomenti delle relazioni, infatti:

* è possibile definire relazioni a partire da fatti, vincoli, eccetera;
* l’esecuzione funziona per differenti combinazioni di input e output.

Si può quindi descrivere la risoluzione di problemi combinatori come quelli NP-Completi utilizzando un programma logico.

Questo approccio può comportare una risoluzione di tipo generate & test che verte verso l’esplosione quadratica per problemi anche piccoli.

## Programmazione a vincoli su domini finiti (CLP)

La CLP è un tipo di programmazione in cui le variabili prendono valori su un dominio definito dal programmatore, nel caso della tricolorabilità di un grafo ad esempio, ogni variabile è un noto e ognuna può assumere uno tra tre valori, ossia i tre possibili colori.

In questo tipo di programmazione, il risoluzione di vincoli viene interrogato per indicare se esiste o no una soluzione.

Quando il risolutore dà esito negativo per il problema di tricolorabilità? Quando un nodo non può essere colorato, quindi quando l’insieme di possibili valori è un insieme vuoto.

## Answer Set Programming (ASP)

L’ASP è un tipo di programmazione nata dalla semantica per la negazione, qui il problema e le condizioni sono rappresentati come nella programmazione logica, aggiungendo costrutti per rendere la descrizioni più concisa.

La soluzione non è conseguenza logica della descrizione, bensì di un suo modello stabile.

La descrizione è di tipo generate-define-test:

* generate perchè descrive le scelte che generano le soluzioni candidate;
* define perchè definisce altre relazioni in base a quelle scelte;
* test impone le condizioni che caratterizzano una soluzione.

Il procedimento di soluzione utilizza l’inferenza proposizionale, ciò permette di ridurre l’esplosione combinatoria.

I meccanismi di un risolutore ASP sono in parte condivisi con i risolutori per il problema NP-Completo SAT.

In ogni casi, si può esprimere ogni problema in NP, l’idea infatti è quella di sviluppare dei solver efficienti per le singole classi di problemi su cui mappare tutti gli altri.

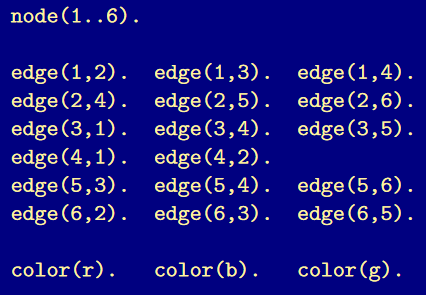
Rispetto a SAT, il linguaggio di rappresentazione di ASP permette anche variabili non booleane e la negazione per default, le quali permettono rappresentazioni più concise.

ASP si può quindi considerare un vero e proprio linguaggio per rappresentare la conoscenza.

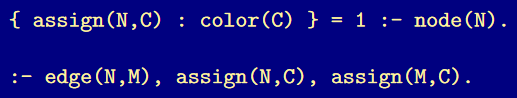
## Colorabilità di un grafo

Il problema di colorabilità di un grafo indica se è possibile assegnare a ogni nodo un colore (su tre disponibili) in modo che due colori uguali non siano mai adiacenti.

Questo problema si può rappresentare utilizzando i seguenti predicati:

* node è un predicato unario che indica i nodi;
* edge è un predicato binario che indica gli archi;
* color è un predicato unario che indica il colore.

Considerando il grafo a destra, l’insieme di predicati nella foto sotto indica l’istanza del problema equivalente, infatti i nodi vanno da 1 a 6, tutti gli archi sono quelli visibili sul grafo e ci sono tre colori.

Cosa manca? Ebbene manca il cosiddetto encoding del problema, cioè quella parte che permette di trovarne una soluzione.

Nel caso del graph coloring, l’enconding è la funzione di assegnamento di un colore a un nodo, visibile nell’ultima foto a destra.

## Come funziona il processo di risoluzione ASP?

Il processo di risoluzione ASP si rappresenta come nella foto a destra.

A partire da un problema, si ottiene una programma logico attraverso un processo detto modeling.

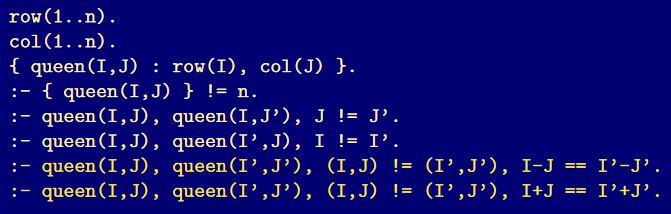
Un grounder permette di ricavare una versione del programma logico che non utilizza variabili. Nel caso del graph coloring ad esempio, il grounder produce un’insieme di regole formato dall’istanza del problema più un’estensione dell’encoding in cui è presente ogni combinazione di valori.

L’output del grounder viene poi dato al solver il quale crea i cosiddetti modelli stabili, dei modelli particolari in cui le regole applicate non vanno tra di loro in conflitto.

## Problema delle n regine

Il problema delle n regine indica come posizionare n regine in una scacchiera n\*n in modo che non siano sotto scacco tra loro.

Qui l’istanza del problema è data dai predicati unari row e col, indicanti rispettivamente righe e colonne, e dal predicato binario queen, la posizione di una regina.

Inoltre, si inserisce una regola per evitare di inserire più regine del necessario.

L’encoding del problema invece è formato da tutte quelle regole che permette di evitare alle regine di essere sotto scacco, quindi tutte le posizioni in cui non possono esserci regine.

## Programmi logici normali

Un programma logico P è un insieme di regole finito su un insieme a di atomi, una regola normale è nella seguente forma:



Nella regola sopra, a0 è la testa della regola mentre l’altra parte è il corpo, quest’ultimo è diviso in una parte positiva, cioè quella con tutti gli atomi positivi, e una negativa.

L’insieme di atomi di un programma è dato dall’unione di tutti gli atomi di tutte le regole.

Un programma p é positivo se è formato da soli atomi positivo.

## Definizione formale

Un set di atomi X è chiuso su un programma P positivo se in ogni regola la testa è in X mentre il body è un suo sottoinsieme.

Visto come una formula, X corrisponde a un modello di P.

Il più piccolo set di atomi chiuso su un programma P positivo è denotato da Cn(P), esso è un modello stabile di P.

Considerando le clausole di Horn, un set di clausole definite corrisponde al più piccolo e unico modello.

Questo modello è la semantica intesa di un set di clausole.

## Idea di base

Si considera la seguente formula:



Con premesse negative, esistono tre possibili modelli che la soddisfano ({p,q}, {q,r} e {p,q,r}), tra questi però uno solo è stabile, cioè {p,q}. L’unico modello stabile è uno dei modelli classici del programma P, tutti i suoi atomi sono giustificati da molte regole.

Informalmente, un set di atomi X è un modello stabile di un programma logico P se è uno dei modelli classici oppure se tutti gli atomi sono giustificati da qualche regola in P.

La riduzione P^X è un’istanza del programma P in cui si considerano solo gli atomi del set X, essa è definita come segue:

  
Da ciò si può concludere che X è un modello stabile di P se X è il più piccolo modello di P^X, cioè Cn(P^X)=X.

## Qual è l’idea del modello stabile?

L’idea del modello stabile è quella di considerare veri sono quelli che sono veri nella testa della regola, ipotizzando quindi che tutto ciò è vero per fare sparire gli atomi negativi.

Nel caso dei programmi positivi, questi hanno un unico modello minimo se il programma è composto da clausole di Horn.

Un programma logico può avere da zero a molti modelli stabili, se un set X è un modello stabile del programma P, allora è un modello di P visto come formula.

Allo stesso modo se X e Y sono modelli stabili di un programma P normale, allora X e Y sono disgiunti.

## Programmi con variabili

Tutte le cose dette in precedenza valgono anche in generale, rispetto a prima la differenza è che le regole considerano le istanze ground, cioè quelle ottenute rimpiazzando tutte le variabili in una regola r con altri termini.



Nella formula sopra, var(r) indica il set di tutte le variabili che occorrono in r, Θ invece è la sostituzione.

Un’istanza di un programma ground Pg è l’unione di tutte le regole ground di un programma P ottenute tramite groundizzazione.

Questa fase può essere fatta in modo intelligente riducendo le istanze ground.

Una regola è sicura se ogni sua variabile occorre in modo positivo nei letterali del body, quindi un programma è sicuro se ha tutte regole safe.

Questa limitazione è utile per limitare il ground, infatti si cerca sempre di mettere nella premessa delle condizioni per tutte le variabili.

## Vincoli di integrità

Un vincolo di integrità serve per eliminare le soluzioni candidate inaspettate, cioè quelle che non rispettano la definizione del problema.

Nel caso della tricolorabilità, una soluzione inaspettata può essere formata da due nodi vicini con lo stesso colore assegnato.

Un vincolo di integrità si rappresenta con ← compilando solo la parte destra della regola.



Esiste anche una versione embedded della regola avente la seguente notazione:



Nella regola sopra, x è un nuovo simbolo non appartenente all’insieme A di atomi, in essa:

* x falso dà luogo a x vero e ciò non dovrebbe succedere;
* x vero invece non soddisfa la formula.

Tutto ciò per dire che i solver accettano anche cose del genere.

## Choice rules

Una choice rule serve a indicare le possibili soluzioni candidate, essa si indica con ← mettendo a sinistra le soluzione.



L’idea di queste regole è quella di considerare il programma prima con e poi senza queste regole, le soluzioni candidate possono quindi essere vere cos’ come non esserlo e quindi essere o meno nel modello stabile.

Anche queste regole presentano una forma più compatta, infatti si pul dividere la regola sopra in un insieme di 2m+1 regole, una per ogni letterale a sinistra più una per riunirle tutte.

## Regole di cardinalità

Le regole di cardinalità controllano la cardinalità più bassa di un sottoinsieme, come se ci fosse una sorta di contatore interno.



Il valore di I è un lower bound, quindi si deve intendere come >= I.

Per ottenere l’upper bound, si ragiona allo stesso modo ma mettendo il numero a destra rispetto alle graffe.

Combinando le due cose, si possono regole regole di cardinalità che valgono per un dato intervallo.



Questi vincoli possono essere utilizzati anche nelle teste delle regole.

## Letterali condizionali

I letterali condizioni sono paricolari letterali nella forma I:I1,...,In, essi sono rappresentabili come una lista di elementi in un set {I|I1,...,In}.

L’espansione dei letterali condizionali dipende dal contesto.

## Statement di ottimizzazione

Gli statement di ottimizzazione esprimono funzioni di costo multiple per minimizzare o massimizzare qualcosa.





Uno statement di ottimizzazione è una direttiva che istruisce il risolutore ASP a calcolare il modello stabile ottimale minimizzando/massimizzazione una somma pesata di elementi.

Oltre a esse è possibile utilizzare funzioni di aggregazione per ottenere un singolo valori a partire da un insieme di input.



Nella formula sopra:

* alpha indica la funzione di mapping da Z a Z unito a (meno) infinito;
* la freccia indica una relazione tra i due insiemi citati prima;
* k è un elemento di Z;
* ai sono gli atomi;
* wi sono interi.

## operatore di conseguenza

Considerando un programma P positivo e un insieme di atomi X, l’operatore di conseguenza Tp è definito come segue:



Tp X è l’insieme di tutte le teste delle regole in cui i letterali del corpo sono contenuti nell’insieme X.

Di questo operatore ne esiste anche una versione iterata in cui si applica Tp tante volte quanto richiesto.

Questo operatore ha le seguenti proprietà:

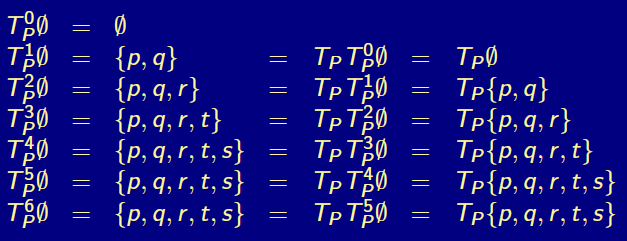
* Cn(P) è l’unione per i>0 di tutti i Tip applicati su un insieme vuoto;
* se X è contenuto in Y, allora TpX è contenuto in TpY;
* Cn(P) è il punto fisso più piccolo di Tp.

### Esempio

Dato il seguente programma P:



Si applica Tp tante volte, ottenendo i seguenti risultati:



T6p applicato all’insieme vuoto corrisponde a Cn(P), cioè {p,q,r,t,s}.

## Approssimare modelli stabili

L’idea è quella di approssimare un modello stabile X a partire da due set di atomi L e U tali che L è contenuto in X e quest’ultimo in U.

I set L e U costituiscono quindi i limiti inferiori e superiori del set X.

Dato che X è un modello stabile di un programma P se e solo se X=Cn(P^X), si può osservare che P^Y è contenuto in P^X e quindi Cn(P^Y) è in Cn(P^X).

Sono inoltre presenti le seguenti proprietà:

* se L è contenuto in X, allora X è contenuto in Cn(P^L);
* se X è contenuto in U, allora X contiene Cn(P^U);
* se le prima due regole valgono, allora X contiene L unito a Cn(P^U) ed è contenuto in U intersecato con Cn(P^L).

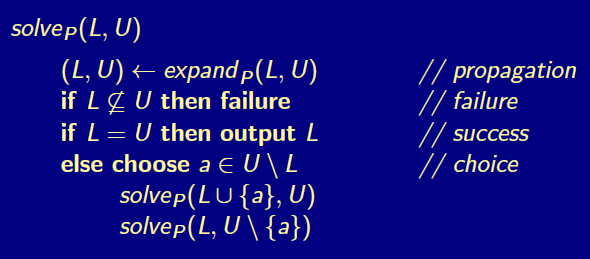
Una seconda idea consiste nel rimpiazzare L e U rispettivamente con L unito a Cn(P^U) e U intersecato con Cn(P^L), si continua così fino a quando non ci sono più cambiamenti.

si può osservare che a ogni iterazione L diventa più grande mentre U si rimpicciolisce, inoltre:

* un set X che contiene L ed è contenuto in U è invariante per ognuno modello stabile di P;
* se L non è contenuto in U, P non ha modelli stabili;
* se L e U sono uguali, allora L è un modello stabile di P.

## C’è un modo semplice per farlo?

sì ed è il seguente algoritmo:



Questo approccio è molto vicino a quello effettuato dal solver ASP smodels, ispirato dalla procedura DPLL:

* si costruisce un albero attraverso una ricerca in backtracking;
* un nodo di quest’albero corrisponde a un’interpretazione a tre valori;
* lo spazio di ricerca viene poi ridotto derivando conseguenze deterministiche, risolvendo i conflitti e facendo una scelta basata su euristiche.