**RAGIONAMENTO PROPOSIZIONALE**

**E INFERENZA**

Le affermazioni sul mondo forniscono vincoli su ciò che potrebbe essere vero. I vincoli possono essere specificati [**estensionalmente**](https://artint.info/2e/html/ArtInt2e.Ch4.S1.SS2.html) come tabelle di assegnazioni legali a variabili, o [**intensionalmente**](https://artint.info/2e/html/ArtInt2e.Ch4.S1.SS2.html) in termini di formule.

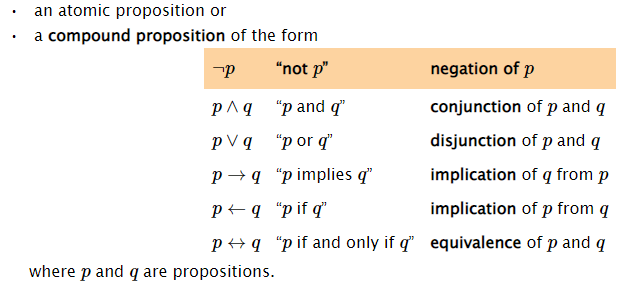
Le proposizioni permettono di specificare vincoli e query con i seguenti benefici:

1. è più conciso e leggibile fornire un'affermazione logica sulla relazione tra alcune variabili piuttosto che utilizzare una rappresentazione estensionale.
2. La forma della conoscenza può essere sfruttata per rendere più efficiente il ragionamento.
3. Sono modulari, quindi piccole modifiche al problema si traducono in piccole modifiche alla base di conoscenza. 🡪 “debug semplice”
4. Il tipo di domande a cui un agente può dover rispondere può essere più ricco delle singole assegnazioni di valori alle variabili.

**SINTASSI DEL CALCOLO PROPOSIZIONALE:**

Una **proposizione** è una frase, scritta in un linguaggio, che ha un valore di verità in un mondo. Una proposizione è costruita da proposizioni atomiche usando connettivi logici. Una **proposizione atomica** , o **atomo** , è un [simbolo](https://artint.info/2e/html/ArtInt2e.Ch4.S1.SS1.html) . Usiamo la convenzione che le proposizioni consistono di lettere, cifre e il carattere di sottolineatura (\_) e iniziano con una lettera minuscola.

Le proposizioni possono essere costruite da proposizioni più semplici usando connettivi logici. Una **proposizione** o **una formula logica** è



Gli operatori ¬, ∧, ∨, →, ← e ↔ sono **connettivi logici** . E l’ordine in cui sono stati presentati fornisce il loro ordine di precedenza. Nei casi di incertezza è possibile usare le parentesi come segue:



Con uso di parentesi diventa:

A picture containing text

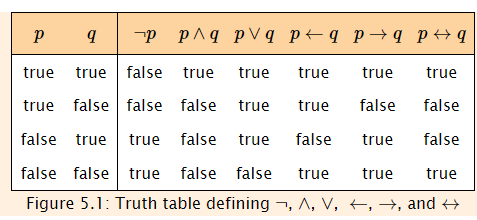
Description automatically generated

**SEMANTICA DEL CALCOLO PROPOSIZIONALE:**

La semantica specifica come mettere in corrispondenza i simboli della lingua con il mondo.  Intuitivamente, gli atomi hanno un significato per qualcuno e sono veri o falsi nelle interpretazioni. La verità degli atomi dà il valore di verità delle altre proposizioni nelle interpretazioni.

Un **interpretazione** consiste di una funzione π  la quale mappa gli atomi nell’insieme {true, false}. Se π(a) = true, allora l’atomo a è **true** nell’ interpretazione. Se π(a) = false, allora l’atomo a è **false** nell’ interpretazione. Possiamo pensare a π come l’insieme degli atomi che sono veri, per cui quelli che non fanno parte di tale insieme sono falsi.

Chiaramente per poter derivare il valore di verità di una proposizione ci affidiamo alle tavole di verità:



Una **base di conoscenza** è un insieme di proposizioni dichiarate vere e un elemento della base di conoscenza è detto **assioma,** prende il nome di assioma poiché essendo presente nella KB è dato per vero.

Un **modello** di una base di conoscenza KB è un'interpretazione in cui tutte le proposizioni presenti nella KB sono vere.

Se KB è una base di conoscenza e g  è una proposizione, diciamo che g è una conseguenza logica della KB e lo scriviamo come segue:



Se g è vera in ogni modello della KB, altrimenti diremo che g non è conseguenza logica della KB ovvero quando ci sarà un modello in cui g è falsa e lo scriveremo come segue:



Se KB ⊧ g diciamo anche che g **segue logicamente** dalla KB , o che la KB **implica** g.

**VISIONE UMANA DELLA SEMANTICA:**

La descrizione della semantica non ci dice perché la semantica è interessante o come può essere usata come base per costruire sistemi intelligenti. L'idea alla base dell'uso della logica è che, quando un **progettista di basi di conoscenza** ha un mondo particolare da caratterizzare, il progettista può scegliere quel mondo come **interpretazione prevista**  e quindi andare a scegliere significati per i simboli rispetto a quel mondo e scrivere proposizioni su ciò che è vero in quel mondo. Quando il sistema calcola una conseguenza logica di una base di conoscenza, il progettista può interpretare questa risposta rispetto all'interpretazione prevista poiché conosce la semantica associata ai simboli presenti nella KB.

La metodologia utilizzata da un progettista di basi di conoscenza per rappresentare un mondo può essere espressa come segue:

**Passo 1**

Un progettista di knowledge base sceglie un dominio di attività o un mondo da rappresentare, che è l'interpretazione prevista o detta interpretazione intesa.

**Passo 2**

Il progettista della base di conoscenza seleziona gli atomi per rappresentare le proposizioni di interesse. Ogni atomo ha un significato preciso rispetto all'interpretazione voluta.

**Passo 3**

Il progettista della base di conoscenza **indica** le proposizioni del sistema che sono vere nell'interpretazione prevista. Questo è spesso chiamato **assiomatizzazione del dominio** , dove le proposizioni date sono gli **assiomi** del dominio.

**Passo 4**

Il progettista della knowledge base può ora **porre** domande sull'interpretazione prevista ed Il sistema può rispondere a queste domande.

Una specificazione del significato dei simboli è chiamata **ontologia** . Le ontologie possono essere specificate in modo informale nei commenti, ma sono sempre più specificate nei linguaggi formali per consentire l'interoperabilità semantica, ovvero la capacità di utilizzare insieme simboli provenienti da diverse basi di conoscenza. Tale significato deve essere noto a chi utilizzerà la KB.

**VISIONE DEL COMPUTER SULLA SEMANTICA:**

Il progettista della base di conoscenza che fornisce informazioni al sistema ha un'interpretazione prevista e interpreta i simboli in base a tale interpretazione prevista. Il progettista afferma la conoscenza, in termini di proposizioni, su ciò che è vero nell'interpretazione prevista. Il computer non ha accesso all'interpretazione prevista, ma solo alle proposizioni nella base di conoscenza.

 L'interpretazione prevista è un modello degli assiomi se il progettista della base di conoscenza è stato veritiero secondo il significato assegnato ai simboli.

Supponiamo che la KB rappresenti la conoscenza dell'interpretazione prevista; ovvero, l'interpretazione prevista è un modello della KB e questo è tutto ciò che il sistema conosce sull'interpretazione prevista. Se KB ⊧ g, allora g deve essere vero nell'interpretazione prevista, perché è vero in tutti i modelli della base di conoscenza. Se KB ⊧̸ g, ovvero g non è una conseguenza logica di KB , esiste un modello di KB in cui g è falso. Per quanto riguarda il computer, l'interpretazione prevista potrebbe essere il modello di KB in cui g è falso, e quindi non sa se g è vero nell'interpretazione prevista.

*Il computer non conosce il significato dei simboli, ma può comunque trarre delle conclusioni in base alla conoscenza presente nella KB.*

È molto importante capire che, finché non consideriamo i computer con la percezione e la capacità di agire nel mondo, il computer non conosce il significato dei simboli. È l'umano che dà significato ai simboli. Tutto ciò che il computer sa del mondo è quello che gli viene detto del mondo. Tuttavia, poiché il computer può fornire conseguenze logiche della base di conoscenza, può trarre conclusioni che sono vere nel mondo.

**VINCOLI PROPOSIZIONALI:**

Le formule logiche forniscono una forma concisa di vincoli con una struttura la quale dà vita alla classe di problemi di soddisfacibilità proposizionale.

Un problema di soddisfacibilità proposizionale è definito come segue:

Una variabile booleana è una variabile con dominio {true, false}. Se X è una variabile booleana, possiamo scrivere X = true come la variabile scritta in minuscolo, x, e scrivere  X = false come  ¬x.

**[ Una clausola è una disgiunzione di atomi che possono essere sia negativi che positivi]**

Una  **clausola** è una espressione della forma l1 ∨ l2 ∨… ∨ lk, dove ogni li è un letterale. Un letterale può essere o un atomo positivo o la sua negazione; per cui un letterale è l’assegnazione di un valore ad una variabile. Una clausola è soddisfatta in un mondo possibile se e soltanto se almeno un letterale della clausola è vero nel mondo possibile.

Una clausola rappresenta un vincolo su di un insieme di variabili booleane.

È possibile convertire qualsiasi CSP finito in un problema di soddisfacibilità proposizionale:

Un variabile Y di un CSP con dominio { v1 , … , vk } può essere convertita in k variabili booleane  { Y1, …, Yk } dove la variabile booleana Yi è true quando Y ha valore vi è invece false altrimenti, ovvero se ha uno degli altri valori del dominio della variabile del CSP. Ogni Yi  è detta **variabile indicatrice**. Per ogni una di tali variabili booleane indicatrici è possibile associare un atomo (o letterale come visto nella descrizione di un problema di soddisfacibilità proposizionale) e per tanto avremo k atomi  y1, …, yk usati per rappresentare il CSP di partenza. Inoltre, dovrà essere imposto un vincolo su queste variabili poiché mentre nel CSP la Y prendeva un solo valore del dominio ne consegue che solo uno dei k atomi nel problema di soddisfacibilità proposizionale potrà essere vero in un dato momento e gli altri dovranno essere falsi e questo vincolo può essere scritto come segue:

¬yi ∨ ¬yj for i < j and y1 ∨ … ∨ yk

Infine, ci sono clausole per ogni assegnazione a false in ogni vincolo, le quali specificano le assegnazioni illecite.

Nel capitolo inerente ai CSP avevamo visto l’algoritmo GAC basato sulla consistenza degli archi, adesso possiamo renderlo più efficiente utilizzandolo su problemi di soddisfacibilità proposizionale.

L’algoritmo della consistenza degli archi può essere adoperato per potare sia il dominio delle variabile che l’insieme dei vincoli nei quali le variabili potate sono coinvolte.

Se ad X è assegnato vero allora tutte le clausole con X = true diventano ridondanti, questo significa che saranno automaticamente soddisfatte e perciò potranno essere rimosse.

Se ad X è assegnato vero, allora in qualsiasi clausola in cui X = false potremmo semplificare la clausola andando a rimuovere X = false. Questo passo è detto **Unit Resolution.**

Quindi se dopo la potature delle clausole come l’abbiamo descritta c’è una clausola che contiene una sola assegnazione Y=v, allora gli altri valori possono essere rimossi dal dominio di Y. Se tutte le assegnazioni sono rimosse da una clausola allora il vincolo è insoddisfacibile e non c’è soluzione.

Text, application

Description automatically generated

Se una variabile ha lo stesso valore in tutte le clausole rimanenti e l’algoritmo deve solamente trovare un modello allora può assegnare il valore che compare nelle clausole alla variabile. Ovvero se una variabile Y appare solo come  Y=true ovvero non è presente ¬y in nessuna clausola (per cui le clausole rimanenti sono tutte y ∨ x ∨ b) allora è possibile assegnare tale valore ad Y. Una variabile che ha un solo valore in tutte le clausole e detta **pure literal (letterale puro).**

L’algoritmo **DPLL(Davis-Putnam-Logemann-Loveland)** è un algoritmo di ricerca esaustiva basato sul backtracking il quale implementa pruning del dominio a dei vincoli e domain splitting e assegna i letterali puri ove possibile.

La struttura proposizionale dei problemi (in particolare dei CSP) permette di rendere più efficiente la ricerca locale per i seguenti motivi:

* È possibile assegnare solo tra 2 valori vero o falso per cui l’algoritmo non perderà tempo nella ricerca nei domini per assegnare un valore.
* Cambiando un valore in una clausola non soddisfatta la renderà soddisfatta.
* Se una variabile viene modificata in modo che diventi vera, solo le clausole in cui appare negativamente possono diventare insoddisfatte e tutte le clausole in cui appare positivamente devono diventare soddisfatte.

**CLAUSOLE PROPOSIZIONALI DEFINITE**

Il linguaggio delle clausole definite proposizionali è un sotto linguaggio del calcolo proposizionale, il quale permette di non avere incertezze o ambiguità.

**SINTASSI DELLE CLAUSOLE DEFINITE PROPOSIZIONALI:**

* Una proposizione atomica o semplicemente atomo è equivalente al atomo del calcolo proposizionale.
* Una clausola definita ha la seguente forma: h ← a1 ∧ … ∧ am, dove h è un atomo e prende il nome di testa “**head”** della clausola e ogni ai è un atomo e possiamo leggere tale clausola definita come “h se a1 and …. and am”. Inoltre, se m > 0 allora la clausola definita prende il nome di **rule (regola)**  e i congiunti a1 ∧ … ∧ am prendono il nome di **body (corpo).** Se invece m = 0 allora rimane solo la testa e tale clausola definita prende il nome di **fact (fatto).**

**Per cui con questo nuovo sotto linguaggio del calcolo proposizionale ovvero delle clausole definite proposizionali una KB risulta essere un insieme di clausole definite.**

Una clausola definita h ← a1 ∧ … ∧ am, è falsa in un interpretazione I se a1 … am ovvero gli atomi del corpo sono tutti veri in I e h la testa risulta false in I, altrimenti in tutti gli altri casi la clausola definita sarà vera.

Le clausole definite risultano essere una forma restrittiva di clausole infatti :

a ← b ∧ c ∧ d è equivalente alla clausola : a ∨ ¬b ∨ ¬c ∨ ¬d.

quindi una clausola definita risulta essere una particolare clausola nella quale vi è un solo letterale positivo e i restanti sono negativi.

L’obbiettivo di descrivere un mondo tramite clausole definite e quindi per mezzo di una KB contenente l’interpretazione che vogliamo dare di questo mondo è quello di determinare quali fatti ( o situazioni ) riguardanti il mondo sono vere oltre a quelle già presenti nella KB. Per cui una volta rappresentato il dominio di un problema potremo chiedere ad un agente intelligente di verificare se alcune proposizioni risultano essere conseguenza logica della KB datagli e per fare ciò porremo al sistema delle **query**, sotto forma di :   
ask b. dove b è un atomo o una congiunzione di atomi. La query sottoposta al sistema avrà una risposta “**yes**” nel caso in cui risulti essere una conseguenza logica della KB, oppure “**no**” nel caso non lo sia e in quest’ultimo caso non si intende che la proposizione chiesta o l’atomo sia falso nell’interpretazione intesa, ma soltanto che non è possibile determinare il suo grado di verità data l’attuale KB.

**PROOFS:**

Ora che sappiamo cosa significa una risposta per una query vediamo come essa venga calcolata dal sistema, in particolare come il sistema dimostra la query tramite il processo di **deduzione,** il quale risulta essere una specifica forma di **inferenza** e questo risulta essere l’**inference engine** dell’agente.

Facciamo distinzione tra i seguenti termini:

* **Proofs (dimostrazione) :** è una dimostrazione derivabile meccanicamente a riguardo di una proposizione che segue logicamente da una KB.
* **THEOREM (teorema) :** è una proposizione dimostrabile (e quindi che è possibile dimostrare che sia conseguenza logica della KB)
* **Proof Procedure (procedura di dimostrazione o algoritmo dimostrativo):**  è un algoritmo che potrebbe essere non deterministico il quale è capace di derivare le conseguenze logiche dalla KB attraverso un processo di deduzione.

 KB ⊢ g è usato per indicare la proof procedure che è capace di provare che g sia conseguenza logica della KB ovvero capace di provare che KB ⊧ g.

Una proof procedure è detta **sound** “sana” se tutto quello che può essere derivato dalla KB risulta essere una conseguenza logica della KB ovvero in formule se  **KB ⊢ g** allora **KB ⊧ g.**

Mentre è detta  **complete** “completa” quando c’è una dimostrazione per ogni conseguenza logica della KB ovvero in formule se **KB ⊧ g** allora **KB ⊢ g.**

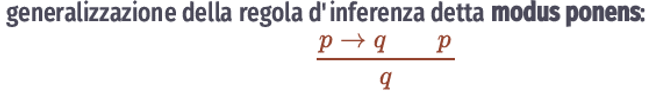
**PROOF PROCEDURE BU & TD**

Presentiamo di seguito le 2 proof procedure utilizzate dagli agenti intelligenti per effettuare deduzione dalla KB fornita.

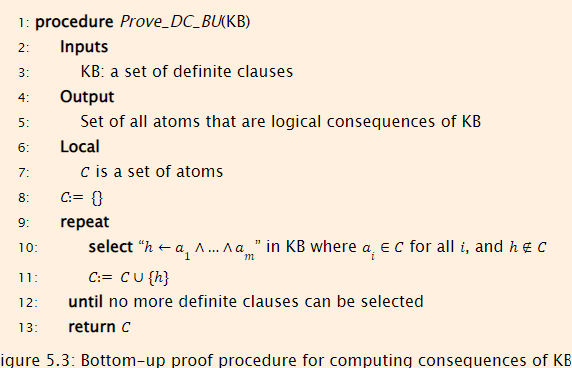
**BOTTOM-UP PROOF PROCEDURE 🡪 BU:**

Questa procedura di dimostrazione costruisce l’inferenza sugli atomi che sono già stati stabiliti essere veri nell’interpretazione intesa e prosegue andando a dimostrare gli altri atomi che servono per completare la dimostrazione di una query. Questa procedura è detta **forward chaining.**  L’algoritmo si basa su una regola di derivazione nota come **modus ponens**:

questo si permette di dire che avendo la clausola definita h ← a1 ∧ … ∧ am, presente nella KB , se ogni ai è stato derivato e quindi provato come vero allora anche h può essere derivata come vera.



La regola di inferenza del modus pones ci dice che, se sappiamo che p 🡪 q è vera e sappiamo anche che p è vera allora possiamo dedurre (o inferire) che q è vera poiché il vero implica solamente il vero.



L’insieme C è l’insieme di tutte le conseguenze logiche che possono essere derivate dalla KB.

Vediamo un esempio dell’algoritmo Bottom-up in azione:

Table

Description automatically generated with medium confidence

La procedura del BU ha le seguenti proprietà:

* **Soundeness:** supponiamo che ci sia un atomo in C che non sia conseguenza logica della KB, e supponiamo che tale atomo sia h. Supponiamo quindi che I è l’interpretazione nella quale h è falso. Ora se l’algoritmo ha generato h in C è perché deve esserci stata una clausola definita nella KB del tipo h ← a1 ∧ … ∧ am e ciò significa che a1 ∧ … ∧ am sono già stati dimostrati come veri e quindi aggiunti a C stando al funzionamento descritto del BU. Ma questo ci dice che tale clausola ha un corpo di letterali tutti veri e la testa falsa per cui è falsa, ma se tale clausola è nella KB risulta vera e questa è una contraddizione per cui non può essere che in C ci sia un atomo che non sia conseguenza logica della KB.
* **Complexity:** l’algoritmo cicla con un massimo che è dato dal numero di clausole definite presenti nella KB, per cui la complessità è lineare rispetto alle dimensioni della KB.
* **Fixed Point:** l’insieme C delle conseguenze è detto punto fisso poiché applicando altre regole di derivazione a quelle già usate non sarà possibile trovare un numero di conseguenze inferiore a quelle trovate in C.
* **Completeness:** Se supponiamo **KB ⊧ g ,** allora **g** è vera in ogni modello della **KB** e quindi è vera nel modello minimale, è vera in C ed è vera nella procedura di dimostrazione ovvero **KB ⊢ g.**

**TOP-DOWN PROOF PROCEDURE 🡪 TD:**

Questa procedura di dimostrazione a differenza della precedente lavora “**backward**” ovvero parte dalla query e cerca di risolvere ogni atomo del corpo per poter dimostrare la query, di fatto appena viene sottoposta una query al sistema con tale query il sistema costruirà la **clausola di risposta iniziale** del tipo : **yes 🡨 query.**

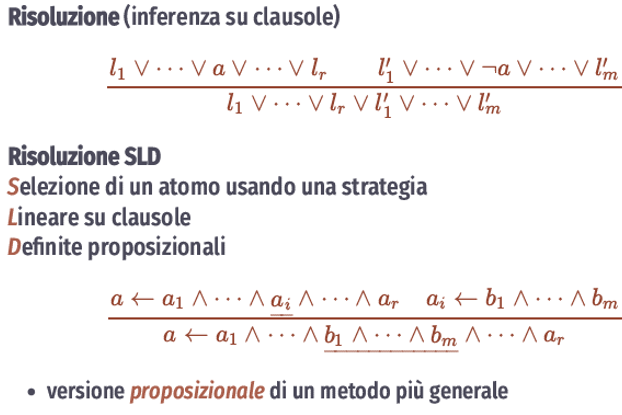
**Ex**

**𝖺𝗌𝗄  a1∧ … ∧am**

Diventerà trasformata dal sistema una clausola di risposta del tipo:

**yes ← a1∧ … ∧am**

Una volta creata tale clausola iniziale potrà avere inizio la procedura dimostrativa detta “**propositional definite clause resolution**” o anche conosciuta come “**SLD resolution**”. In questo algoritmo viene selezionato un atomo del corpo della clausola di risposta iniziale e una volta selezionato tale atomo prende il nome di **subgoal**, l’obiettivo dell’algoritmo sarà quello di provare che ogni atomo del corpo è vero tramite il processo di inferenza logica chiamato **risoluzione**.



Il processo di **RISOLUZIONE** si adotta su qualsiasi tipo di clausola che risulti essere una disgiunzione di letterali. In particolare, si prendono in esame 2 disgiunzioni di letterali nelle quali in una vi è un letterale a in forma positiva e nell’altra un letterale not a ovvero in forma negativa, allora se entrambe le clausole risultano vere esse non dipendono dal valore di verità dell’atomo a in questione e quindi nel processo di risoluzione quell’atomo può essere non considerato.

Nell’algoritmo TD ci sono 2 passi che richiedono attenzione il primo è la **selezione** di un atomo del corpo della query di risposta, in questa fase non ha importanza quale che sia l’atomo scelto poiché per dimostrare la query bisognerà provare tutti gli atomi del corpo per cui questa è una scelta **“don’t care”**, poiché non ha importanza da quale inizio, al contrario il passo detto di **scelta** ovvero quello che va a selezionare dalla KB un assioma presente che permette di risolvere un subgoal una volta selezionato tale subgoal è una scelta che può influenzare notevolmente sia la riuscita dell’algoritmo TD sia la sua efficienza. Di fatto ci sono delle sequenze di scelte che potrebbero portare l’algoritmo a fallire quando scegliendo in modo diverso sarebbe riuscito a provare la query, non solo nei casi in cui non sia possibile dimostrare una query poiché essa non è conseguenza logica della KB è bene avere come **scelta** la clausola definita che permette di far fallire l’algoritmo il più velocemente possibile.

Quando durante gli step dell’algoritmo selezioniamo la clausola definita della KB che ci permette di dimostrare un subgoal si dice che stiamo scegliendo il **risolvente** di tale atomo e questo comporterà una modifica della clausola di risposta iniziale ove l’atomo selezionato sarà sostituito con il corpo del risolvente scelto.

**Ex**: supponiamo di avere il seguente risolvente per il subgoal a1 della precedente clausola di scelto tale risolvente il corpo del risolvente sarà sostituito al posto di a1 nella clausola di risposta iniziale:

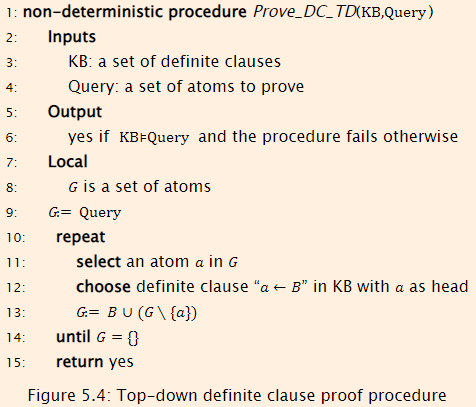
**yes ← b1 ∧ … ∧ bp ∧ a2 ∧ … ∧ am**

Il processo termina quando si ha **yes ← ,** ovvero quando il corpo della clausola di risposta rimane vuoto e la testa è yes.

Una **derivazione SLD**  rispetto ad una data query risulta essere una sequenza di clausole di risposta del tipo : ɣ0 , ɣ1, … , ɣn dove abbiamo che:

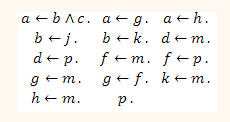
* ɣ0 è la clausola iniziale di risposta
* ɣi è il risolvente per ɣi – 1
* ɣn è la risposta che termina l’algoritmo

L’algoritmo TD può essere pensato come ad un insieme di atomi G che necessitano di essere provati tramite le clausole definite presenti nella KB, inizialmente G contiene tutti gli atomi presenti nella query sottoposta al sistema successivamente per ogni subgoal un atomo selezionato viene ad essere sostituito dal corpo degli atomi risolventi e tali atomi saranno inseriti in G al posto del atomo subgoal selezionato. Quando G risulta vuoto TD giunge ad una conclusione.

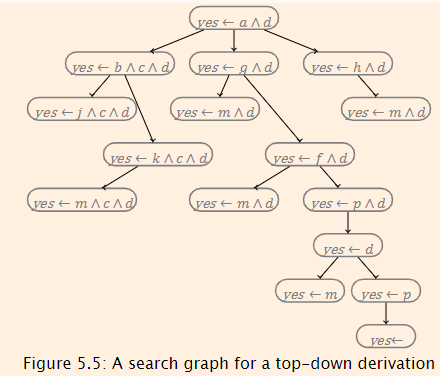


Abbiamo visto che l’algoritmo TD è non deterministico e non dà garanzia di terminare infatti la terminazione è influenzata dalle scelte che facciamo nei processi di risoluzione, tuttavia mettendo assieme l’algoritmo TD e una strategia di selezione idonea a non far fallire l’algoritmo possiamo vedere il problema come una ricerca in un grafo i cui nodi sono le clausole di risposta generate dall’algoritmo TD. I vicini di un nodo risultano essere tutte le possibili risoluzioni del subgoal (ovvero dell’atomo selezionato) infine il nodo “yes 🡨” rappresenterà il goal della ricerca.

**Ex:** data la seguente KB con le sue clausole definite

****

E ponendo al sistema come domanda ovvero come query **ask a ∧ d,** avremo ilseguente grafo di ricerca assumendo che sia sempre selezionato come nuovo subgoal l’atomo più a sx nel corpo:



Il grafo in questo problema sarà generato dinamicamente fornendo il meccanismo di scelta delle clausole definite presenti nella KB in modo da poterle usare per generare i vicini di un nodo, su tale grafo possiamo adottare una strategie di ricerca tra tutte quelle che conosciamo ove chiaramente potremmo optare per una che dia garanzia nel ritrovare una soluzione quando essa esiste e che non debba essere necessariamente ottimale poiché il nostro scopo è solamente quello di provare la query.

Comparando i 2 algoritmi abbiamo che la procedura di dimostrazione TD può impiegare più tempo a riprovare lo stesso atomo più volte, mentre la procedura BU dimostra ogni atomo solo una volta. Tuttavia, la procedura BU dimostra ogni atomo (ovvero tutti quelli presenti nella KB), mentre la procedura TD dimostra solo gli atomi rilevanti per la query.

Nell’algoritmo TD se non dovesse essere presente una strategia di potatura dei cicli è consigliato come strategia di selezione degli atomi del corpo che rappresentano i subgoal di selezionare quello che porterà più velocemente al fallimento della procedura di dimostrazione.

**BACKGROUND KNOWLEDGE E OSSERVAZIONI:**

Un osservazione rappresenta un informazione che può essere ricevuta online dagli utenti, da sensori oppure da fonti di conoscenza. Queste osservazioni sono un insieme di proposizioni atomiche, le quali possono essere aggiunte alla KB oppure essere trattate separatamente come avviene per ( l’abduzione o per il ragionamento probabilistico).

**QUERING THE USER [interrogare l’utente]:**

In alcuni sistemi intelligenti si ha la rappresentazione del mondo sotto forma di clausole definte in una KB, ma non si posseggono informazioni inerenti a un caso specifico che può verificarsi nel mondo che abbiamo rappresentato, in tale situazione occorre un meccanismo che permetta all’agente intelligenti di recuperare tali informazioni inerenti ad uno specifico caso per farlo abbiamo bisogno di un meccanismo detto **ask-the-user** il quale permetterà di recuperare informazioni sui casi specifici direttamente dall’utente che utilizza il sistema intelligente. Nel meccanismo **ask-the-user** abbiamo degli atomi che prendono il nome di **askable**  essi sono degli atomi sui quali la KB non ha clausole che permettano di provare il grado di verità di tali atomi, ma l’utente può conoscerne il grado di verità a runtime.

Con tale meccanismo possiamo suddividere gli atomi in:

* atomi di cui l’utente non conosce il grado di verità detti **never askable**
* atomi **askable**, possono essere chiesti all’utente e il grado di verità fornito viene registrato
* atomi **askable**, che sono già stati chiesti e non devono essere richiesti

Con tale meccanismo il processo di dimostrazione vede l'utente porre una domanda al sistema, e ad ogni passaggio il sistema pone una domanda, a cui si risponde trovando le clausole definite rilevanti o chiedendo all'utente. L'intera interazione può essere caratterizzata da un protocollo di domande e risposte tra due agenti, l'utente e il sistema.

**SPIEGAZIONE A LIVELLO DI CONOSCENZA:**

Affinché un sistema intelligente sia usabile esso non può semplicemente fornire risposte ad una query posta e aspettarsi o sperare che l’utente vi creda, serve invece un meccanismo di spiegazione della risposta data in modo da convincere l’utente sul perché sia stata data tale risposta. Questo meccanismo di spiegazione è implementato tramite 3 tipi di domande da porre al sistema:

1. **how question** 🡪 spiega come una risposta è stata fornita
2. **why question** 🡪 usata per chiedere al sistema spiegazione a riguardo di una domanda posta dal sistema in particolare per venire a conoscenza del grado di verità di un atomo askable.
3. **whynot question** 🡪 usato per chiedere al sistema di dare spiegazione sulla prova non riuscita di un atomo.

Queste 3 domande possono essere usate per ripercorrere i passi della dimostrazione condotta dal sistema.

Una domanda **why** è utile perché:

* Vogliamo che il sistema appaia intelligente, trasparente e affidabile. Sapere perché è stata posta una domanda aumenterà la fiducia dell'utente nel fatto che il sistema funzioni in modo sensato.
* Una delle principali misure di complessità di un sistema interattivo è il numero di domande poste a un utente, che dovrebbe essere ridotto al minimo. Sapere perché è stata posta una domanda aiuterà il progettista della conoscenza a ridurre questa complessità.
* Una domanda irrilevante di solito è un sintomo di un problema più profondo.
* L'utente può imparare qualcosa dal sistema sapendo perché il sistema sta facendo qualcosa. Questo apprendimento è molto simile a un apprendista che chiede a un maestro perché il maestro sta facendo qualcosa.

**DEBUGGING DI UNA KB:**

Le spiegazione fornite dal sistema all’utente non solo fanno sì che il sistema guadagni la fiducia dell’utente e sia credibile, ma permettono all’ingegnere della conoscenza di debuggare la KB in caso di risposte errate o non previste.

Il knoledge-level debugging è un processo che permette di trovare gli errori nella KB inerenti solo al significato dei simboli e al quello che risulta vero nel mondo rappresentato.

Gli errori che possono sorgere durante il processo di debug sono di 4 tipi:

1. Una risposta incorretta è stata data, ovvero un atomo che risulta falso nell’interpretazione intesa è stato derivato come vero.
2. Una risposta che ci si aspettava non viene data dal sistema, ovvero una procedura di dimostrazione fallisce quando non avrebbe dovuto fallire.
3. Si genera un ciclo infinito
4. Il sistema pone domande all’utente irrilevanti per la dimostrazione della query posta

**INCORRECT ASWER [risposta errata] 🡪 falso-positivo**

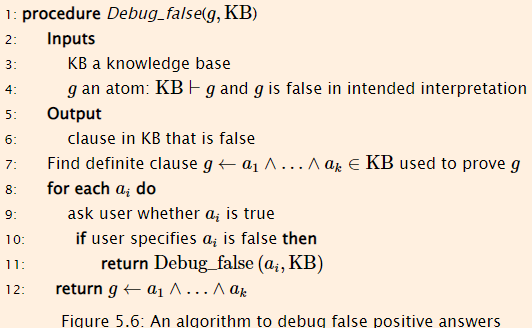
Una risposta incorretta è anche detta **false-positive error**  e tale errore è possibile solo nel caso di una procedura sound che utilizzi una clausola definita presente nella KB che sia incorretta. Per debuggare una risposta incorretta un esperto del dominio necessita solo rispondere a domande del tipo si/no.

**Ex:**

Supponiamo che ci sia un atomo g che è stato dimostrato(quindi è dato per vero), ma è falso nell'interpretazione prevista. Allora ci deve essere una regola g ← a1 ∧ … ∧ ak nella base di conoscenza utilizzata per dimostrare g. Quindi o

* uno degli ai è falso nell'interpretazione prevista, nel qual caso può essere sottoposto a debug nello stesso modo, o
* tutti gli ai sono veri nell'interpretazione prevista; in questo caso, la clausola definita g ← a1 ∧ … ∧ ak deve essere errata.

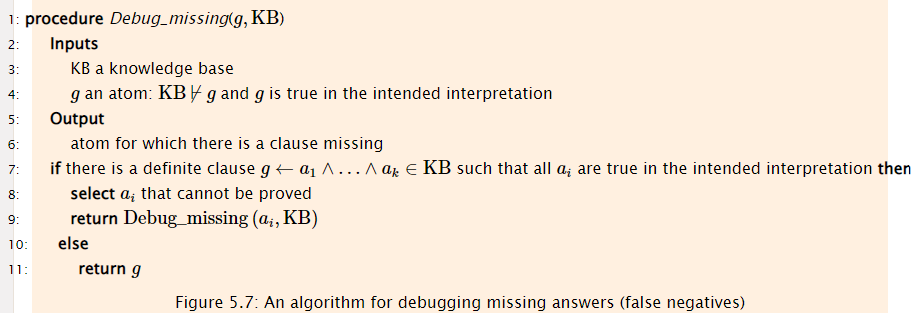
Questo meccanismo porta ad un algoritmo capace di debuggare i falsi positivi e quindi di risolvere gli errori di risposte incorrette, detto **DEBUG\_FALSE(g, KB):**

****

L’algoritmo ritorna come risultato le clausole errate presenti nella KB ovvero quelle date per vere poiché presenti nella KB, ma che in realtà risultano false nell’interpretazione intesa.

**MISSING ASWERS [risposte mancati, ovvero risposte che ci si aspettava il sistema desse per vere]:**

In questa tipologia di errore un atomo che è vero nel dominio(ovvero vero nell’interpretazione intesa) non viene derivato come tale dalla KB, in questo caso parliamo di **false-negative error**. Questa volta l’algoritmo che useremo per il debug è differente infatti nel caso di un incorrettezza (ovvero falso-positivo) l’algoritmo ci restituiva clausole errate nella KB, qui il problema è differente abbiamo una mancanza di conoscenza che non permette di derivare determinate queries per cui con questo algoritmo **debug\_missing(g, KB)**  andremo a trovare quali sono le clausole definite mancanti nella KB tali per cui non si è potuto derivare un atomo.



Se supponiamo che g sia vero, ma non viene derivato come tale dall’algoritmo allora poiché la procedura di dimostrazione ha fallito per g significa che tutte le clausole definite presenti nella KB le quali hanno come testa g hanno i corpi contenenti atomi che falliscono e per tali ragioni non si può dimostrare g. Ora se supponiamo che in una di questa clausole definite che vedono g in testa c’è ne sia una che avrebbe dovuto garantire la dimostrazione ciò vuol dire che gli atomi che costituiscono il corpo di tale clausola definita sono tutti veri nell’interpretazione intesa, ma poiché il corpo fallisce dato che g è fallito ci deve essere necessariamente un atomo di tale corpo che fallisce e questo atomo sarà vero nell’interpretazione intesa, ma falso stando al ragionamento dell’agente usando la KB e per ciò potremmo ricorsivamente ritrovare tutti questi atomi che risultano veri nell’interpretazione ma che non sono derivati, alla fine del processo se tali atomi non possono essere derivati poiché non ci sono le clausole definite che permettano la loro derivazione allora tali clausole dovranno essere aggiunte alla KB.

In questo algoritmo di debug della KB prende un posto importante la domanda **whynot** che può essere integrata nell’algoritmo per chiedere al sistema di fornire spiegazione sul perché un dato atomo non è stato provato.

**INFINITE LOOPS [cicli indefiniti]**

Questo problema sorge solo nel momento in cui la KB considerata è **ciclica**, una KB ciclica presenta clausole definite ove la testa di una clausola definita compare come corpo in altre clausole definite ove le teste di queste ultime risultano essere il corpo della prima clausola definita presa in esame ovvero:

**a ← … a1 …**

**a1 ← … a2 …  
 ……**

**……**

**an ← … a …**

dove nel caso in cui n = 0 avremmo avuto una singola clausola definita nella KB del tipo

**a ← a**

Chiaramente in questa condizione la dimostrazione non termina e continua a ciclare indefinitamente sostituendo gli atomi presenti nei corpi con le clausole cicliche.

Per evitare tali problemi e debuggare KB cicliche può essere usata la procedura TD con l’aggiunta di una lista detta “**lista degli antenati**” la quale terrà traccia degli atomi presenti nel corpo di una clausola che dovrà essere provata.

**DIMOSTRAZIONE PER CONTRADDIZIONE:**

Il ragionamento su clausole definite può essere utilizzato per implementare dimostrazioni per contraddizione permettendo tuttavia l’uso di regole che danno vita alla contraddizione. Per permettere tali regole avremo bisogno di espandere il linguaggio delle clausole definte andandole a sostituire con le **clausole di Horn**, le quali permettono di definire **vincoli di integrità:**

**false ← a1 ∧ … ∧ ak**

dove false è un atomo speciale che è falso in tutte le interpretazioni. Per cui una **clausola di Horn sarà o una clausola definita oppure un vincolo di integrità.**

**Equivalenza logica**  
false ← a1 ∧ … ∧ ak **⬄** ¬a1 ∨ … ∨ ¬ak

Questo permette di provare che una congiunzione di atomi risulta falsa in una data interpretazione, ne consegue che una KB fatta di clausole di Horne è tale da implicare[dimostrare] la negazione degli atomi.

Vediamo un esempio di una KB con clausole di Horne:

**false ← a ∧ b**

**a ← c**

**b ← c**

Da questa KB possiamo dedurre che **c** è falso, infatti, se ci fosse vero nell’interpretazione intesa allora **a 🡨 c** , **b 🡨 c** risulterebbero ambe due vero poiché è presenti nella KB e aventi l’antecedente o corpo vero, tuttavia ciò vorrebbe dire che **a** e **b**  risultano entrambi veri e ciò contraddirebbe **false 🡨 a ^ b** , il cui grado di verità è vero poiché presente nella KB ma ciò è impossibile poiché il V implica il F è F (V 🡪 F = F). Per tale ragione abbiamo dimostrato che **c**  è falso mentre **¬c** è vero. Allora diciamo che:

**KB ⊧ ¬c**ovvero che **¬c**  è conseguenza logica della KB.

Infine, diciamo che un insieme di clausole è **insoddisfacibile** se non ha un modello ovvero se esiste un Interpretazione tale da far coesistere le clausole. Un insieme di clausole è detto **inconsistente** se tramite una procedura di dimostrazione è possibile derivare l’atomo speciale **false**. Inoltre, diremo che per una procedura di dimostrazione che risulta essere  **sound e complete**  un insieme di clausole è inconsistente se e solo se è insodisfacibile.

Avendo introdotto le clausole di Horne e potendo derivare come conseguenza logica della KB atomi falsi è possibile usare questo nuovo ragionamento in sistemi diagnostici intelligenti dove l’agente supporrà il grado di verità sugli oggetti del mondo il quale potrebbe essere contradetto da uno stato di inconsistenza della KB, per cui l’agente ragionerà sul mondo supponendo la verità se tali supposizioni sono inconsistenti ne cambierà il grado di verità.

Un **assumibile** è un atomo il cui grado di verità può essere assunto per effettuare una procedura di dimostrazione per contraddizione e tale procedura andrà a derivare una disgiunzione di assumibili negati.

Un **conflitto** per una KB fatta di clausole di Horne è un insieme di assumibili che portano a derivare l’atomo speciale false.   
C = {c1 , … , cr} è un insieme di conflitti per KB se KB ∪ {c1, …, cr} ⊧ false.

E in questo caso una risposta sarà data da : KB ⊧ ¬c1 ∨ … ∨ ¬cr

**CONSISTENCY-BASED DIAGNOSIS:**

Fare assunzioni riguardo a quello che normalmente funziona nel mondo e in seguito derivare tramite dimostrazione quali possono essere i possibili componenti anormali è la base del ragionamento usato per la **DIAGNOSI BASATA SU CONSISTENZA.**

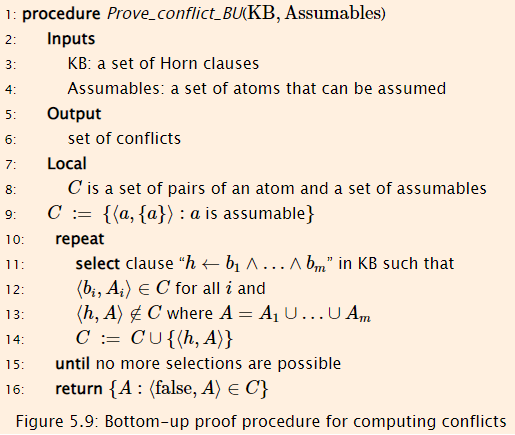
In questa situazione il guasto di un componente (supponendo di ragionare in un sistema intelligente per la diagnostica) può essere individuato assumendo un insieme di guasti come assumibili e andando a vedere quali inconsistenze essi portano alla KB.

Dato un insieme di conflitti, una **diagnosi basata sulla consistenza** è un insieme di assumibili che ha almeno un elemento in ogni conflitto. Una **diagnosi minima** è una diagnosi tale che nessun sottoinsieme è anche una diagnosi. Per una delle diagnosi, tutti i suoi elementi devono essere falsi nel mondo che viene modellato.

Alla luce del concetto di assumibili e delle clausole di Horn andiamo a vedere come si comportano gli algoritmi di dimostrazione BU e TD per KB basate su clausole di Horn.

**BU WITH HORN CLAUSES:**

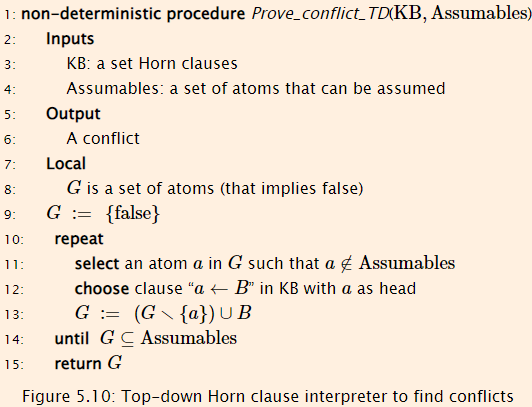
In questa versione dell’algoritmo vengono restituite coppie <a, A> dove a è un atomo e A è un insieme di assumibili che portano a derivare a come conseguenza logica della KB.



Questo algoritmo è anche chiamato **assumption-based truth maintenance system ATMS**.

**TD WITH HORN CLAUSES:**

In questa versione le modifiche apportate all’algoritmo TD prevedono la query di risposta iniziale settata a false ovvero data una query nel TD il primo passo detto prima clausola di risposta prevedeva la creazione della clausola definita **yes 🡨 query**, adesso con le clausole di Horn la prima clausola di risposta è una clausola di Horn del tipo **false 🡨 query.** Oltre a questa modifica gli assumibili usati nella dimostrazione vengono raccolti in un insieme.

****

Le differenti scelte prese possono portare ad avere come risultato un diverso insieme di conflitti o potrebbero portare l’algoritmo a fallire la procedura di dimostrazione.

**ASSUNZIONE DI CONOSCENZA COMPLETA**

L’assunzione di conoscenza completa porta il sistema a affermare che tutto quello che non può essere derivato dalla KB allora è falso. Un esempio di KB con assunzione di conoscenza completa è dato da un **DATABASE**.

L’assunzione di mondo chiuso ovvero di conoscenza completa permette all’agente di ragionare tramite meccanismi di default sullo stato del mondo. L’assunzione di mondo chiuso richiede che tutto quello che sia rilevante a riguardo del mondo sia conosciuto dall’agente intelligente.

Sotto l’ipotesi di mondo chiuso possiamo introdurre il  **completamento di Clark** il quale ci permette di derivare la doppia implicazione tra la testa di una clausola di Horn e i corpi disgiunti di tutte le clausole di Horne presenti nella KB che hanno la stessa testa presa in considerazione ovvero:

a ← b1  
 ⋮

……..

a ← bn

se a è vera in una delle interpretazione allora uno dei bi che figurano come corpo delle precedenti clausole deve essere vero anche esso nell’implementazione intesa dove a è vero e questo significa che possiamo scrivere :

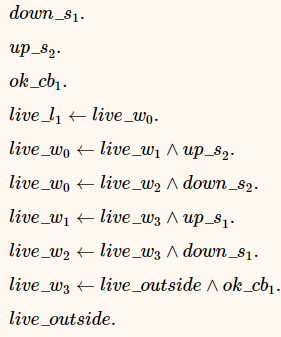
a → b1 ∨ … ∨ bn

tuttavia, le clausole con le quali abbiamo definito a precedentemente ci portano a scrivere:  
a ← b1 ∨ … ∨ bn

ma questo vuol dire che le due formule logiche portano alla seguente equivalenza logica:

a ⇔ b1 ∨ … ∨ bn

**ESEMPIO DI UNA KB CON CLAUSOLE DI HORN A SX CHE VIENE COMPLEMENTATA CON I RISPETTIVI COMPLEMENTI DI CLARK PER OGNI CLAUSOLA PRESENTE:**

 Text, letter

Description automatically generated

Tramite il complemento il linguaggio può essere esteso e infatti è possibile ammettere la negazione degli atomi nei corpi delle clausole. Si usa la tilde per distinguere tale tipo di negazione da quella classica vista nel mondo aperto, la negazione segnalata dalla tilde prende il nome di  **Negazione per fallimento.**

Con il processo di negazione per fallimento un corpo g di una clausola risulta essere una conseguenza logica della KB se KB′ ⊧ g ove KB’ risulta essere il complemento di Clark della KB.

**RAGIONAMENTO NON MONOTONO**:

Una logica risulta essere **monotona** se ogni proposizione che può essere derivata dalla KB può essere anche derivata quando viene ad essere aggiunta conoscenza alla KB e quindi le vengono aggiunte nuove clausole.

Una logica risulta **non-monotona** se alcune conclusioni ovvero proposizioni derivate dalla KB possono essere invalidate ovvero non derivabili nel moment in cui si aggiunge nuova conoscenza. La logica delle clausole definite con negazione per fallimento è una logica non-monotona.

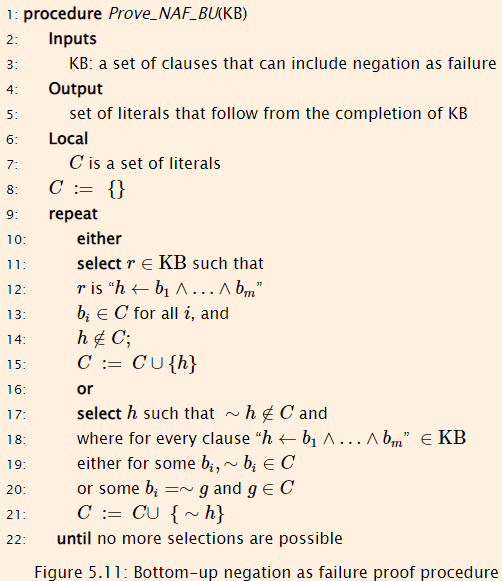
La logica non-monotona è utile per rappresentare situazioni di default e un default risulta essere una regola che può essere usata a meno ché non venga ad essere sovrascritta da una eccezione.

b ← c∧ ∼ ab\_a

In questo esempio vediamo codificata una regola di default che ci dice che b è normalmente vero quado c è vero usando ab\_a anormalità che si verifica rispetto all’atomo a la verità di b può cambiare.

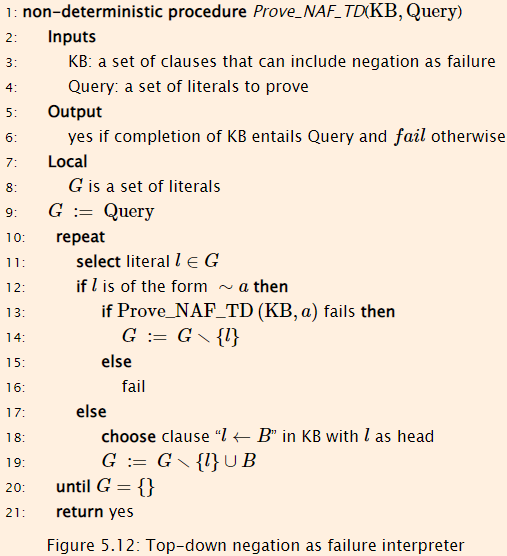
**BUTTOM-UP PROCEDURA PER NEGAZIONE PER FALLIMENTO:**

La differenza con la BU per clausole definite è che all’insieme C delle conseguenze vengono aggiunti tutti i letterali del tipo ∼p quado è possibile determinare che la dimostrazione p fallisce, in questo algoritmo il fallimento nel derivare una conseguenza è definito in maniera ricorsiva ovvero p fallisce quando falliscono tutti i corpi delle clausole che contengono p come testa.



**TOP-DOWN NEGATION AS FAILURE PROCEDURA:**

Quando viene selezionato un atomo negato supponiamo ∼a, una nuova dimostrazione per l’atomo a inizia, se questa fallisce allora ∼a ha successo, viceversa nel caso in cui a ha successo allora l’algoritmo fallisce e deve prende un ‘altra scelta di atomo negato.



**ABDUZIONE:**

L’abduzione è una forma di ragionamento nella quale le assunzioni(sugli atomi) sono fatte per spiegare le osservazioni rilevate sullo stato del mondo. Il termine **abduzione** fu coniato da Peirce per differenziare il questo ragionamento logico da quello per **deduzione** che prevede di determinare quello che logicamente segue da un insieme di assiomi e per differenziarlo dall’**induzione** la quale implica l’inferenza di relazioni generali da esempi visti. Nell’abduzione si ipotizza quello che potrebbe essere vero rispetto ad un fatto osservato e si cerca di determinare se tale ipotesi supporta la verità dell’osservazione.

Per formalizzare l’abduzione utilizzeremo il linguaggio delle clausole di Horn e gli atomi assumibili e invece di aggiungere le osservazione alla KB tale osservazione fatte devono essere spiegate ed una spiegazione per una proposizione g data una KB di clausole di Horn e un insieme di Assumibili è uno scenario nel quale dalla KB facendo uso degli assumibili si può derivare g come conseguenza logica. In formule supposto H l’insieme delle ipotesi tale da essere un sottoinsieme al più uguale ad A insieme degli assumibili allora diciamo che una spiegazione per g è:

* KB ∪ H ⊧ g
* KB ∪ H  false

|  |  |
| --- | --- |
|  |  |

Data tale definizione per una **spiegazione**  su di una preposizione diciamo che una **spiegazione minimale** è il più piccolo insieme H di ipotesi possibile per il quale è possibile derivare la proposizione in questione.

Determinare quello che sta succedendo in un sistema basato su osservazioni è un problema di diagnosi o riconoscimento. Nella diagnosi abduttiva una agente ipotizza un malfunzionamento del sistema ed ipotizza che altre parti del sistema stiano funzionando adeguatamente per dare una spiegazione delle osservazioni fatte. Questa procedura di diagnosi abduttiva differisce dalla diagnosi basata sulla consistenza infatti:

* Nella diagnosi basata su consistenza le ipotesi sono assumibili che riguardano il normale funzionamento, mentre nella diagnosi abduttiva gli assumibili possono rappresentare e suppore comportamenti normali allo stesso modo che comportamenti anormali.
* Nella diagnosi abduttiva le osservazione devono essere spiegate [nel modo in cui abbiamo definito cosa si intenda per una spiegazione di una proposizione], nella diagnosi basata su consistenza le osservazioni sono aggiunte alla KB per poi derivare l’atomo speciale false.