

Appunti di Introduzione all’Intelligenza Artificiale Unipi - 2

Parte

Raffaele Apetino

Marzo 2020

Contents

1 Giochi con avversario	4
1.1 Ciclo Pianifica-Agisci-Percepisci	4
1.2 Definizione spazio degli stati	4
1.3 Game Tree	4
1.4 Algoritmo MIN-MAX	5
1.4.1 Come si calcola il valore MIN-MAX?	5
1.4.2 Costo MIN-MAX	6
1.5 Algoritmo MIN-MAX euristico (con orizzonte)	6
1.5.1 Come si calcola il valore MINIMAX euristico (H-MINIMAX)?	7
1.5.2 Esempio H-MIN-MAX in Tic-Tac-Toe	7
1.6 Problemi con MIN-MAX	7
1.7 Ottimizzazione MIN-MAX	7
1.8 Potatura ALFA-BETA	8
1.8.1 Costo potatura ALFA-BETA	9
1.8.2 Ottimizzazioni ALFA-BETA	9
1.9 Giochi Multiplayer	9
1.10 Giochi Complessi	9
2 Problemi di soddisfacimento di vincoli (CSP)	10
2.1 Formulazione di problemi CSP	10
2.1.1 Problema della colorazione di una mappa	10
2.1.2 Problema 8 regine	11
2.2 Strategie per problemi CSP	11
2.3 Ricerca per problemi CSP	11
2.4 Ricerca con backtracking (BT) a profondità limitata	11
2.5 Codice Algoritmo backtracking	12
2.5.1 Select-Unassigned-Variable()	12
2.5.2 Order-Domain-Values()	12
2.5.3 Inference()	12
2.5.4 Backtrack()	13
3 Agenti Logici KB	14
3.1 Approccio dichiarativo VS Approccio procedurale	14
3.2 Tell-Ask	14
3.3 Base di conoscenza o Base di dati?	14
3.4 Formalismi per la rappresentazione della conoscenza	15

4 Agenti Logici: Calcolo proposizionale	15
4.1 Sintassi (regole)	15
4.2 Semantica (significato)	15
4.3 Conseguenza logica	15
4.4 Il mondo del Wumpus	16
4.4.1 Esempio dal mondo del Wumpus	17
4.5 Formule logiche, Validità e Soddisfacibilità	17
4.6 Inferenza per calcolo proposizionale (PROP)	18
4.7 Algoritmo TT-Entails (Model Checking)	18
4.7.1 Esempio TT-Entails	18
4.8 Algoritmo DPLL (Alg. Soddisfacibilità)	18
4.8.1 Forma a clausole	18
4.8.2 Esempio DPLL	19
4.8.3 Miglioramenti DPLL	19
4.9 Algoritmo WALK-SAT	19
4.9.1 Esempio WALK-SAT	20
4.9.2 Analisi WALK-SAT	20
4.9.3 Problemi SAT difficili	20
4.10 Inferenza come deduzione	21
4.10.1 Alcune regole di inferenza	21
4.11 Regola di risoluzione	21
4.11.1 Regola di risoluzione in generale per PROP	21
4.11.2 Esempio grafo di risoluzione	22
4.11.3 Refutazione	22
4.11.4 Osservazioni	23
5 Agenti Logici: Logica del prim'ordine	23
5.1 Il mondo dei blocchi	23
5.2 Concettualizzazioni	24
5.3 FOL	24
5.3.1 Predicati	24
5.3.2 Termini	24
5.3.3 Formule	24
5.3.4 Interpretazione	24
5.4 Sematica Composizionale	25
5.4.1 Semantica Standard VS Semantica Database	25
5.5 Interazione con la KB tramite FOL	25
5.6 Regola di inferenza per \forall	25
5.7 Regola di inferenza per \exists	26
5.8 Riduzione a inferenza proposizionale	26
5.8.1 Teorema di Herbrand	26
5.9 Verso un metodo di risoluzione per il FOL	26
5.9.1 Forma a clausole	26
5.9.2 Esempio di trasformazione (passo passo)	27
5.10 Sostituzione	27
5.11 Unificazione	28
5.11.1 Algoritmo di unificazione	28
5.11.2 Esempi	28
5.12 Metodo di risoluzione per FOL	29
5.12.1 Esempio metodo di risoluzione	29
5.12.2 Problemi	30
5.12.3 Completezza del metodo di risoluzione	30

5.13	Refutazione	30
5.13.1	Esempio di Refutazione	30
5.14	Risoluzione efficiente per FOL	31
5.14.1	Strategie di cancellazione	32
5.14.2	Strategie di restrizione	32
5.14.3	Strategie di ordinamento	33
5.15	Sistemi a regole	34
5.15.1	Regole in avanti e indietro	34
5.16	Programmazione logica (Backward Chaining)	34
5.16.1	Risoluzione SLD	35
5.16.2	Alberi di risoluzione SLD	35
5.16.3	Esempio di albero SLD	35
5.17	Sistemi a regole in avanti (Forward Chaining - FOL_FC_Ask)	36
5.17.1	Esempio di concatenazione in avanti	36
5.17.2	Analisi di FOL_FC_Ask	37

1 Giochi con avversario

Il modello base a cui ci siamo affidati fin'ora è realizzato su ambienti osservabili, deterministici e con utente singolo. Nella prima parte del corso abbiamo visto che esistono gli ambienti multi-agente, in cui un agente deve tenere conto anche delle azioni degli altri agenti che lo circondano. I giochi con avversario¹ si basano su ambienti deterministici multi-agente competitivi, in particolare su un ambiente reso strategico a causa della presenza di un avversario.

Vedremo i problemi di soddisfacimento dei vincoli (CSP) in cui lo stato ha una struttura fatorizzata. Vedremo che nei sistemi basati su conoscenza lo stato è una "base di conoscenza" (KB) a cui rivolgere domande sul mondo rappresentato in un linguaggio espressivo come il PROP (calcolo proposizionale) o FOL (logica del primo ordine).

1.1 Ciclo Pianifica-Agisci-Percepisci

Ci troviamo nel caso in cui abbiamo due agenti che agiscono a turno, si pianifica considerando le possibili risposte dell'avversario e le risposte alla possibile sua risposta e così via. Una volta decisa la mossa migliore da fare, si agisce, si percepisce la mossa dell'avversario e infine si ri-pianifica la mossa. La decisione ottima teorica è definita come la mossa migliore in un gioco con uno spazio di ricerca completamente esplorabile (per trovarla utilizzeremo l'algoritmo MIN-MAX). E' possibile che ci troveremo in situazioni in cui a causa della complessità non sarà possibile eseguire una esplorazione esaustiva dello spazio. Sfrutteremo, inoltre, tecniche di ottimizzazione della ricerca (algoritmo ALFA-BETA).

1.2 Definizione spazio degli stati

Un gioco può essere definito formalmente come una sorta di problema di ricerca con i seguenti elementi:

- Stati: configurazioni del gioco
- Stato iniziale: configurazione iniziale del gioco
- Player(s): a chi tocca eseguire l'azione nello stato s
- Actions(s): mosse legali in s
- Result(s,a): stato risultante dopo aver eseguito la mossa a nello stato s (modello di transizione)
- Terminal-Test(s): determina la fine del gioco, controlla che s sia lo stato terminale (e quindi il gioco finito)
- Utility(s,p): funzione di utilità (chiamata anche funzione obiettivo o funzione pay-off) che restituisce un valore numerico che valuta gli stati terminali del gioco per p (ad esempio somma di punteggi ecc...)

1.3 Game Tree

Lo stato iniziale insieme alle funzioni Actions e Result definiscono l'albero di gioco (Game Tree). È un albero dove ogni nodo è uno stato del gioco e i rami definiscono una mossa. Ogni foglia definisce uno stato terminale del gioco con associato di un valore di utilità definito dalla funzione Utility. Il prossimo esempio che vedremo è il gioco del Tris, i due giocatori si alternano tra MAX che posiziona le X e MIN che posiziona i cerchi, fino a che si arriva ad una delle foglie dell'albero che rappresentano un nodo terminale.

¹due giocatori, turni alterni, a somma zero (se uno vince, l'altro perde)

1.4 Algoritmo MIN-MAX

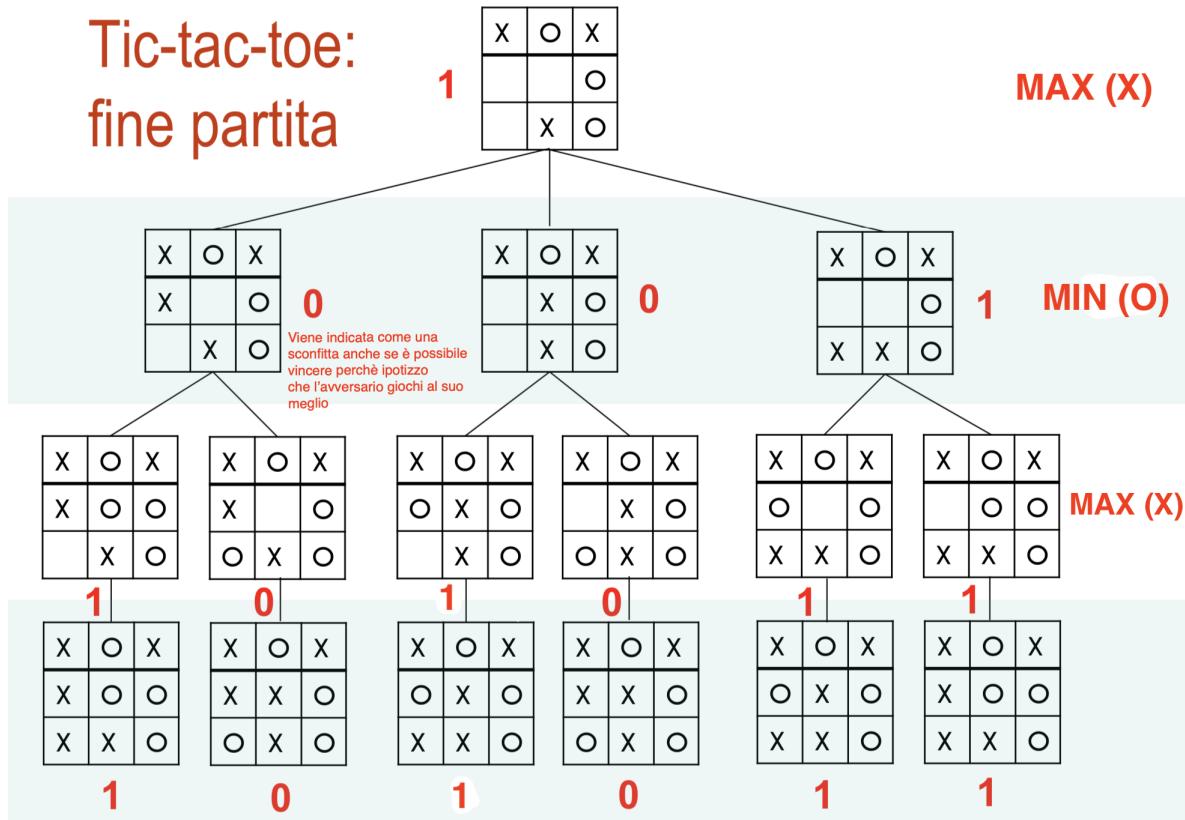


Figure 1: Albero parziale costruito dall'algoritmo MIN-MAX

L'albero creato da MIN-MAX è molto simile ad un albero AND-OR, in cui i nodi MAX hanno il ruolo di OR e MIN quello di AND.

1.4.1 Come si calcola il valore MIN-MAX?

Dato un albero di gioco, la strategia ottima è determinata dal valore "MINIMAX" di ogni nodo:

MINIMAX(s) =

- if (`Terminal-Test(s) == true`) `Utility(s,MAX)`
 - if (`Player(s) == MAX`) $\max_{a \in \text{Actions}(s)}(\text{MINIMAX}(\text{Result}(s, a)))$
 - if (`Player(s) == MIN`) $\min_{a \in \text{Actions}(s)}(\text{MINIMAX}(\text{Result}(s, a)))$

Ovviamente il valore MINIMAX di uno stato terminale è la sua utilità. Altrimenti, per ogni possibile azione (e quindi ramo), MAX preferisce spostarsi sui nodi in cui il valore MINIMAX è massimo, MIN su quelli di valore minimo.

L'albero di gioco conviene esplorarlo in profondità, perché in ampiezza occuperei troppa memoria, ma soprattutto a noi interessa trovare uno dei tanti percorsi di vincita.

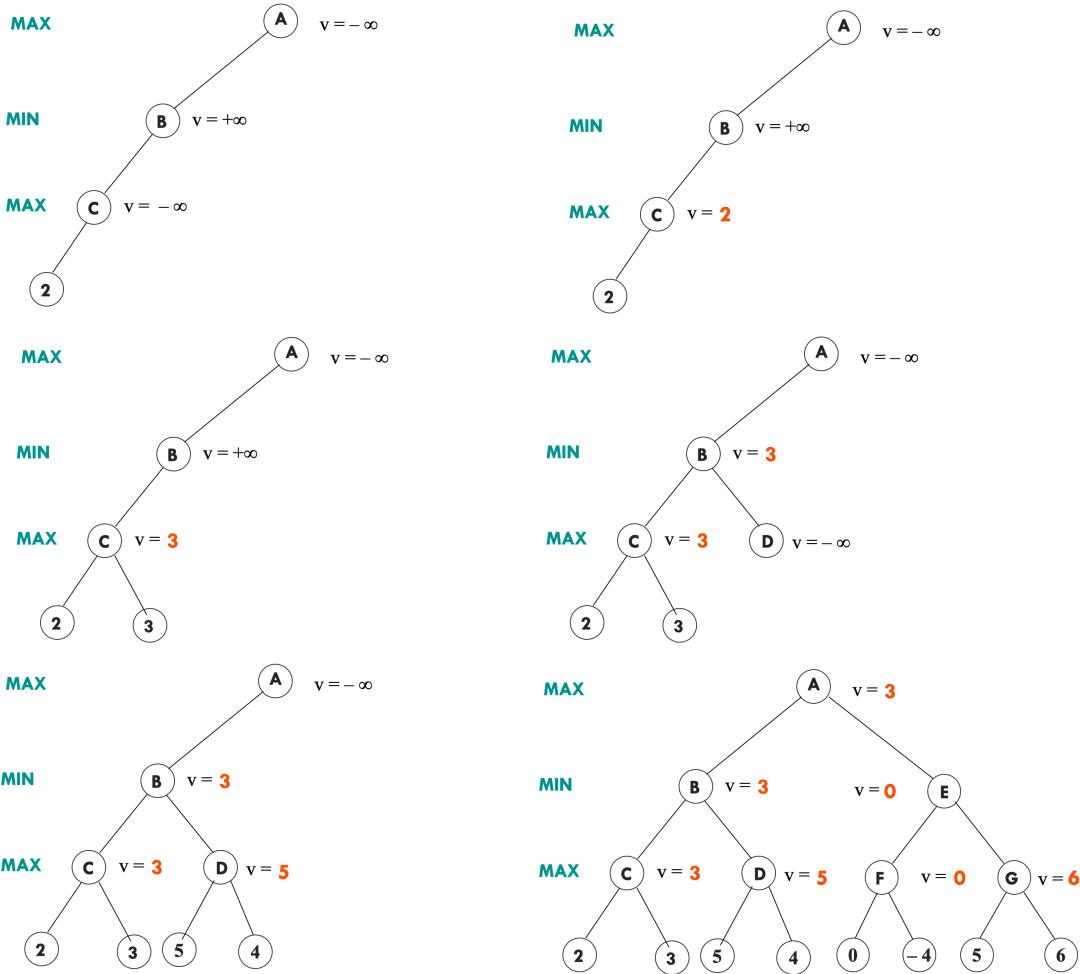


Figure 2: Algoritmo MIN-MAX in azione

1.4.2 Costo MIN-MAX

- Tempo: $O(b^m)$
- Spazio: $O(m)$

In giochi complessi come gli scacchi abbiamo un albero di ricerca molto grande 35^{100} (35 mosse in media, 100 mosse in media per ogni partita quindi 50 mosse per player), purtroppo anche mantenendo solo i nodi distinti, il grafo di ricerca rimane ampio: (10^{40}) nodi! È quindi improponibile una esplorazione sistematica, se non per giochi veramente semplici. Troviamo la necessità di far uso di euristiche per stimare il valore di uno stato del gioco.

1.5 Algoritmo MIN-MAX euristico (con orizzonte)

Nei casi più complessi quindi occorre usare una funzione di valutazione euristica dello stato $\text{Eval}(s)$. La strategia che applicheremo sarà quella di espandere l'albero di ricerca un certo numero di "d" livelli, si valutano gli stati ottenuti e si propaga indietro il risultato con la regola del MAX e MIN. Quindi al posto di eseguire un Terminal-Test eseguiremo un CutOff-Test che ci dirà quando eseguire la funzione di valutazione.

1.5.1 Come si calcola il valore MINIMAX euristico (H-MINIMAX)?

H-MINIMAX(s,d) =

- if (CutOff-Test(s,d) == true) Eval(s)
- if (Player(s) == MAX) $\max_{a \in Actions(s)} H - MINIMAX(Result(s,a), d+1)$
- if (Player(s) == MIN) $\min_{a \in Actions(s)} H - MINIMAX(Result(s,a), d+1)$

1.5.2 Esempio H-MIN-MAX in Tic-Tac-Toe

La funzione di valutazione Eval(s) è una stima dell'utilità attesa a partire da una certa posizione nel gioco. La "qualità" della funzione è fondamentale: deve essere consistente con la vera utilità se applicata a stati terminali del gioco (deve mantenere lo stesso ordinamento dei nodi), deve essere efficiente da calcolare (immaginiamo dei giochi a tempo) ma deve anche essere fortemente correlata alle probabilità di vittoria.

Usiamo come funzione di valutazione la differenza tra le X(s) righe aperte per X e O(s) righe aperte per O. La nostra euristica sarà: $Eval(s) = X(s) - O(s)$

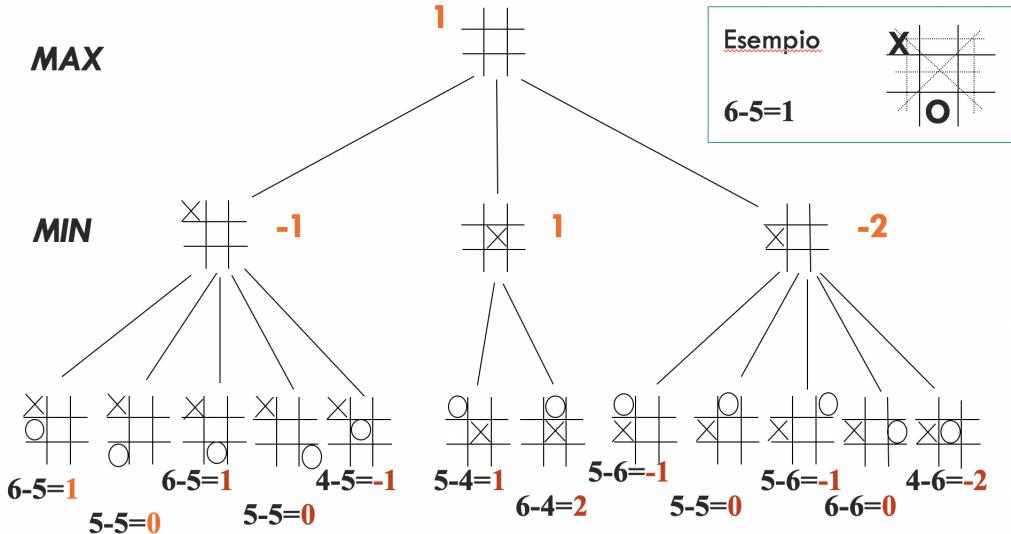


Figure 3: Algoritmo MIN-MAX euristico in azione

1.6 Problemi con MIN-MAX

- Stati non quiescenti: l'esplorazione fino ad un certo livello può mostrare una situazione molto vantaggiosa, ma alla mossa successiva la situazione può peggiorare esponenzialmente. La soluzione da applicare è la funzione di valutazione a stati quiescenti, cioè stati in cui $Eval(s)$ non è soggetto a mutamenti repentini (questo implica la ricerca di quiescenza).
- Effetto orizzonte: può succedere che vengano privilegiate mosse divisorie che hanno il solo effetto di spingere il problema oltre l'orizzonte. Una sorta di serie di mosse che non fanno altro che "scappare" dal problema il più possibile, magari ritrovandosi in un loop infinito.

1.7 Ottimizzazione MIN-MAX

E' necessario esplorare ogni cammino? NO! si può sfruttare un metodo per dimezzare la ricerca pur mantenendo una decisione della prossima mossa corretta (Potatura ALFA-BETA).

1.8 Potatura ALFA-BETA

Il problema principale della ricerca MIN-MAX è il numero di stati che cresce esponenzialmente in relazione alla profondità dell'albero. Purtroppo non possiamo eliminare l'esponente, ma possiamo dimezzarlo.

ALFA-BETA è una tecnica di potatura per ridurre l'esplorazione dello spazio di ricerca in algoritmi MIN-MAX. L'idea è che si vada avanti in profondità fino al livello desiderato, si propaghino indietro i valori e a questo punto si decida se si può abbandonare l'esplorazione nel sotto-albero. Sfruttiamo due valori "MaxValue" (α) e "MinValue" (β) rispettivamente inizializzati a $\alpha = -\infty$ e $\beta = +\infty$. Rappresentano rispettivamente la migliore alternativa per MAX e per MIN fino a quel momento.

I tagli del sottoalbero avvengono quando nel propagare indietro troviamo:

- $v \geq \beta$ per i nodi MAX (taglio β)
 - $v \leq \alpha$ per i nodi MIN (taglio α)

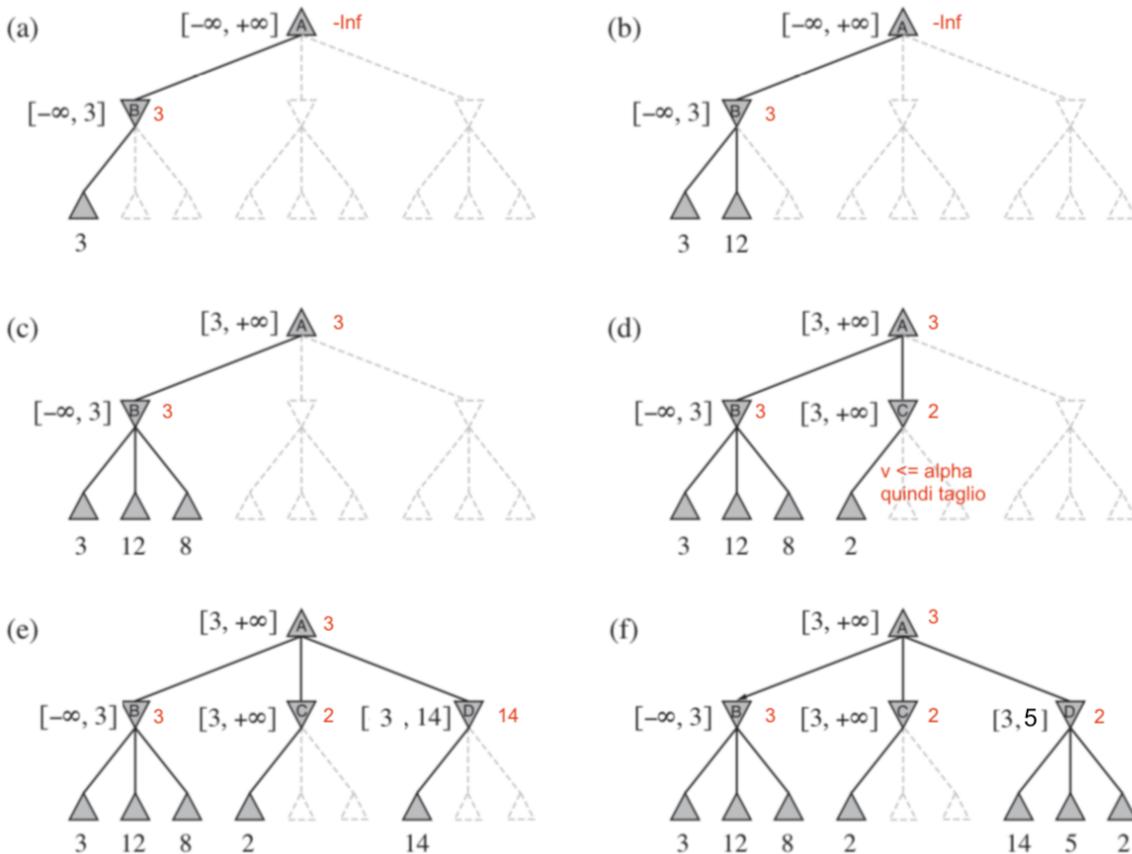


Figure 4: Potatura ALFA-BETA in azione

1.8.1 Costo potatura ALFA-BETA

- Tempo: $O(b^{m/2})$ quindi ALFA-BETA può arrivare a profondità doppia rispetto a MIN-MAX (best case)
- Spazio: $O(m)$

1.8.2 Ottimizzazioni ALFA-BETA

Quale potrebbe essere l'ordinamento ottimale?

- Ordinamento dinamico: usando un approfondimento iterativo si possono scoprire informazioni utili per l'ordinamento delle mosse da usare in una successiva iterazione (mosse killer). Possiamo inoltre tenere in memoria una tabella delle trasposizioni, cioè per ogni stato incontrato si memorizza la sua valutazione.
- Potatura in avanti: si potrebbe anche eseguire una potatura in avanti, cioè esplorare solo alcune mosse ritenute promettenti e tagliare le altre (tagli probabilistici basati su esperienza).
- Database di mosse: possiamo inoltre sfruttare un database di mosse di apertura e chiusura. Nelle prime fasi ci sono poche mosse sensate e ben studiate, inutile esplorarle tutte, mentre per le fasi finali il computer può esplorare off-line in maniera esaustiva e ricordarsi le migliori chiusure.

1.9 Giochi Multiplayer

Molti giochi permettono di far partecipare più di due giocatori. Dobbiamo modificare il singolo valore di ogni nodo con un vettore (ad esempio in foto vengono valutati allo stesso tempo 3 valori, uno per ogni giocatore).

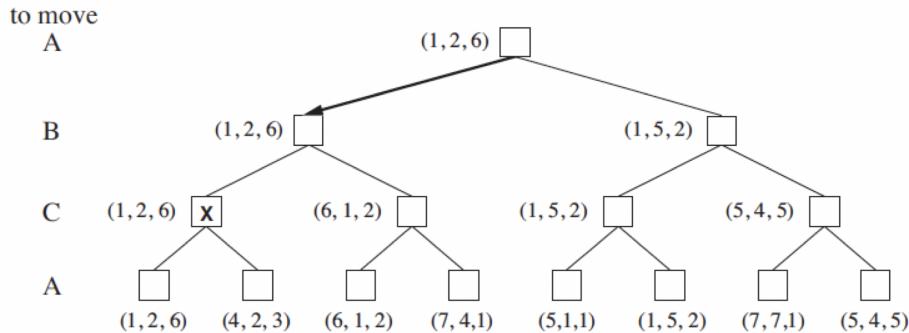


Figure 5: Giochi Multiplayer

1.10 Giochi Complessi

- Giochi stocastici: i giochi in cui è prevista una variabile aleatoria (lancio di dadi oppure la distribuzione di carte)
- Giochi parzialmente osservabili: le mosse sono deterministiche ma non si conoscono gli effetti delle mosse dell'avversario. (Battaglia navale)

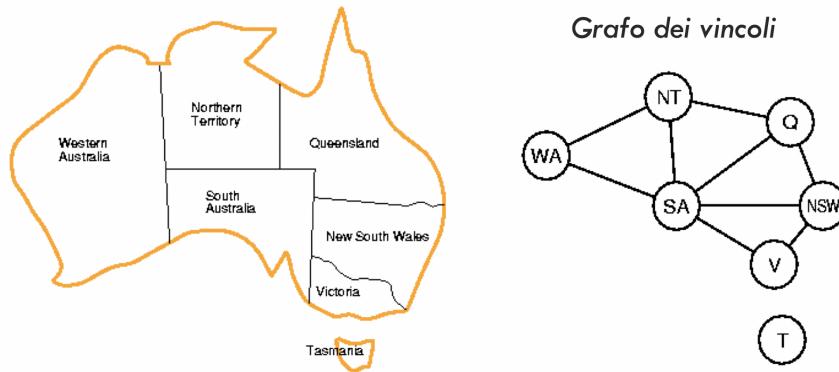
2 Problemi di soddisfacimento di vincoli (CSP)

Sono problemi con una struttura particolare che necessitano algoritmi di ricerca specializzati, si sfrutta una rappresentazione fattorizzata dello stato rendendo esplicita la sua struttura (o almeno in parte). Un problema è risolto quando ogni variabile ha un assegnamento che soddisfa tutti i vincoli. Esistono euristiche generali che si applicano a questi problemi e consentono la risoluzione anche se il problema ha dimensioni significative.

2.1 Formulazione di problemi CSP

- Problema: descritto da tre componenti
 1. X insieme di variabili
 2. D insieme di domini (ogni dominio D_i è a sua volta un insieme di valori possibili per la variabile X_i)
 3. C insieme di vincoli (ogni vincolo C_i è una coppia [variabili,relazione])
- Stato: un assegnamento parziale o completo di valori a variabili (ad esempio $X_i = v_i, X_j = v_j \dots$)
- Stato iniziale: {}
- Azioni: assegnamento di un valore ad una variabile
- Soluzione: un assegnamento completo della formula, dove tutte le variabili hanno un valore consistente, cioè dove tutti i vincoli sono soddisfatti

2.1.1 Problema della colorazione di una mappa



Problema: Si tratta di colorare i diversi paesi sulla mappa con tre colori in modo che paesi adiacenti abbiano colori diversi.

1. $X = \{WA, NT, SA, Q, NSW, V, T\}$
2. $D_{WA} = D_{NT} = D_{SA} = D_Q = D_{NSW} = D_V = D_T = \{\text{red, green, blue}\}$
3. $C = \{WA \neq NT, WA \neq SA, NT \neq Q, NT \neq SA, SA \neq Q, SA \neq NSW, SA \neq V, NSW \neq V, NSW \neq Q\}$

2.1.2 Problema 8 regine

Problema: Si tratta di posizionare le 8 regine sulla scacchiera senza che nessuna sia in grado di mangiarne un'altra.

1. $X = \{Q_1, \dots, Q_8\}$ una regina per colonna della scacchiera
2. $D_i = \{1, 2, 3, 4, 5, 6, 7, 8\}$ numero di riga
3. C = vincoli di non attacco

2.2 Strategie per problemi CSP

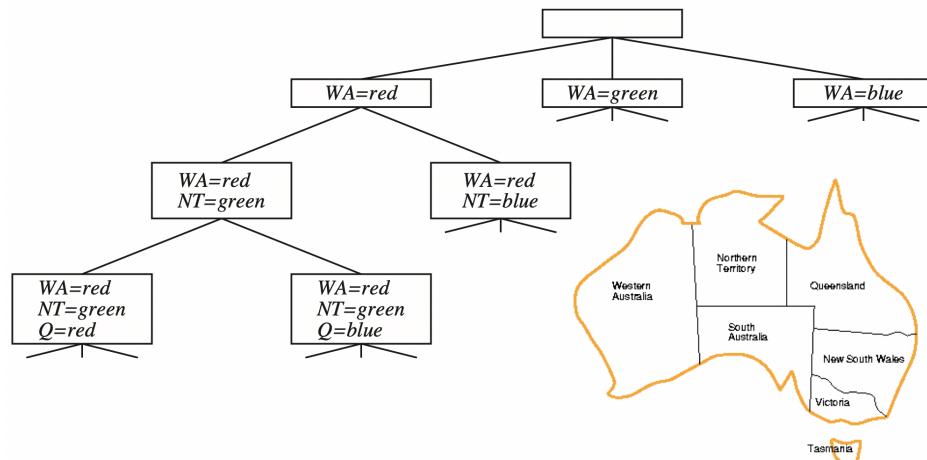
Fin'ora potevamo solo ricercare la soluzione nel grafo degli stati, invece adesso possiamo usare delle euristiche specializzate per questa classe di problemi. Possiamo fare delle inferenze che ci portano a restringere i domini e quindi a limitare la ricerca, in più possiamo eseguire backtracking intelligente.

2.3 Ricerca per problemi CSP

Ad ogni passo si assegna una variabile, la massima profondità della ricerca è fissata dal numero di variabili n. L'ampiezza dello spazio di ricerca invece è definito da $|D_1| * |D_2| * \dots * |D_n|$ (dove $|D_i|$ è il numero di elementi del dominio di X_i). Il fattore di diramazione è pari a $n * d$ al primo passo cioè posso scegliere tra d valori da assegnare a n variabili, al secondo passo $(n - 1) * d$ e così via... si genera così un albero con $n! * d^n$ foglie anche se esistono al massimo d^n assegnamenti completi differenti. Possiamo ridurre drasticamente lo spazio di ricerca sfruttando il fatto che il goal-test è commutativo (non è importante l'ordine con cui scelgo le variabili, quindi ad ogni livello posso scegliere una sola variabile a cui assegnare un valore). Con questa restrizione otteniamo d^n foglie come volevamo.

2.4 Ricerca con backtracking (BT) a profondità limitata

Si esegue un controllo anticipato sulla violazione dei vincoli, perché è inutile andare avanti fino alla fine e poi controllare la correttezza. La ricerca è limitata in profondità dal numero di variabili quindi il metodo è completo. L'algoritmo sceglie ripetutamente una variabile senza assegnamento e a turno prova tutti i valori del dominio di quella variabile fino a trovare una soluzione. Se viene trovata una inconsistenza l'algoritmo si ferma ed esegue il backtracking facendo provare alle chiamate precedenti un nuovo valore.



2.5 Codice Algoritmo backtracking

```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
    return BACKTRACK({ }, csp)
function BACKTRACK(assignment, csp) returns a solution, or failure
    if assignment is complete then return assignment
    var  $\leftarrow$  SELECT-UNASSIGNED-VARIABLE(csp) Quale variabile scegliere?
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do Quali valori scegliere?
        if value is consistent with assignment then controllo anticipato
            add {var = value} to assignment
            inferences  $\leftarrow$  INFERENCE(csp, var, value) Qual è l'influenza di un assegnamento sulle altre variabili? come restringe i domini?
            if inferences  $\neq$  failure then
                add inferences to assignment
                result  $\leftarrow$  BACKTRACK(assignment, csp) Come evitare di ripetere i fallimenti?
                if result  $\neq$  failure then
                    return result
                remove {var = value} and inferences from assignment
            return failure
```

Le parti evidenziate sono i punti dove posso usare euristiche!

2.5.1 Select-Unassigned-Variable()

Scelta delle variabili: qual è la prossima variabile da scegliere?

- MRV (Minimum Remaining Values): scegliere la variabile che ha meno valori legali (residui) da poter assegnare, cioè la variabile più vincolata. Si scoprono prima i fallimenti (fail first)!
- Euristica del grado: scegliere la variabile coinvolta in più vincoli con le altre variabili (la variabile più vincolante o di grado maggiore). (Usata a parità di MRV)

2.5.2 Order-Domain-Values()

Scelta dei valori: una volta scelta la variabile, come scegliere il valore da assegnare?

- Valore meno vincolante: scegliamo il valore che esclude meno valori per le altre variabili direttamente collegate alla variabile scelta. Meglio valutare prima un assegnamento che ha più probabilità di successo, se volessimo tutte le soluzioni l'ordine non sarebbe importante.

2.5.3 Inference()

Propagazione di vincoli: qual è l'influenza di un assegnamento sulle altre variabili?

- Verifica in avanti (Forward Checking o FC): assegnato un valore ad una variabile, si possono eliminare i valori incompatibili per le altre variabili direttamente collegate da vincoli.
- Consistenza di nodo e arco: si restringono i valori dei domini delle variabili tenendo conto dei vincoli unari e binari su tutto il grafo (se una variabile ad un certo punto ha il dominio ridotto all'insieme vuoto, si esegue immediatamente backtracking. Altrimenti si itera finché tutti i nodi ed archi sono consistenti).

2.5.4 Backtrack()

Quando la ricerca arriva ad un assegnamento che viola i vincoli, come posso evitare di ripetere in futuro lo stesso errore?

- Backtracking Cronologico: viene cambiato il valore dell'ultima variabile assegnata, continuando a fallire fino al soddisfacimento dei vincoli.
- Backtracking Intelligente: si considerano alternative solo per le variabili che hanno causato il fallimento (X_i, X_j, \dots), cioè un "insieme dei conflitti". Ad esempio, nel problema delle regine, si parte con tutte le variabili assegnate (tutte le regine sulla scacchiera) e ad ogni passo si modifica l'assegnamento ad una variabile per cui un vincolo è violato (si muove una regina minacciata su una colonna). Questo è un algoritmo di riparazione euristica. Un'altra euristica potrebbe consistere nello scegliere un nuovo valore che crea meno conflitti (euristica dei conflitti minimi).

3 Agenti Logici KB

Vogliamo migliorare le capacità razionali dei nostri agenti dotandoli di rappresentazioni di mondi più complessi. Il mondo è tipicamente complesso, ci serve una rappresentazione parziale e incompleta (una astrazione) del mondo, utile agli scopi dell'agente. Per descrivere ambienti parzialmente osservabili e complessi ci servono linguaggi di rappresentazione della conoscenza più espressivi, ma soprattutto con capacità inferenziali. Gli agenti basati su conoscenza sono dotati di una KB (knowledge base) con conoscenza espressa in maniera esplicita e dichiarativa. La conoscenza può essere codificata a mano, ma anche estratta dai testi o appresa dall'esperienza.

3.1 Approccio dichiarativo VS Approccio procedurale

L'approccio dichiarativo consiste nella creazione di una KB che racchiude tutta la conoscenza necessaria a decidere l'azione da compiere in forma dichiarativa. L'alternativa (approccio procedurale) è scrivere un programma che implementa il processo decisionale, una volta per tutte.

Un agente KB è più flessibile: più semplice acquisire conoscenza in modo incrementale e modificare il comportamento con l'esperienza!

3.2 Tell-Ask

Un agente basato su conoscenza mantiene una base di conoscenza (KB), cioè un insieme di enunciati espressi in un linguaggio di rappresentazione. L'agente interagisce con la KB mediante una interfaccia funzionale definita come Tell-Ask:

- Tell: per aggiungere nuovi enunciati a KB
- Ask: per interrogare la KB
- Retract: per eliminare enunciati

Gli enunciati nella KB rappresentano le credenze dell'agente, le risposte X dell'agente quindi devono essere tali che X è una conseguenza (discende necessariamente) della KB.

Il nostro obiettivo sarà capire quando, avendo una base di conoscenza KB contenente una rappresentazione dei fatti che si ritengono veri, un certo fatto α è vero di conseguenza ($KB \models \alpha$, cioè α è conseguenza logica della KB).

3.3 Base di conoscenza o Base di dati?

Qual è la differenza tra una KB e un database?

- Base di conoscenza: una rappresentazione esplicita dello stato, parziale e compatta, espressa in un linguaggio simbolico, che contiene fatti di tipo specifico e fatti di tipo generale o anche regole. Quello che caratterizza una base di conoscenza è la capacità inferenziale, cioè la capacità di derivare nuovi fatti da quelli memorizzati esplicitamente.
- Base di dati: solo fatti specifici e posso solo interrogarla, non ho modo di inferire nuovi fatti.

Ma il problema fondamentale della rappresentazione della conoscenza sta nel trovare il giusto compromesso tra: espressività del linguaggio di rappresentazione e complessità del meccanismo inferenziale. Sfortunatamente più il linguaggio è espressivo, meno efficiente è il meccanismo inferenziale.

3.4 Formalismi per la rappresentazione della conoscenza

Un formalismo per la rappresentazione della conoscenza ha tre componenti:

1. Una sintassi: cioè un linguaggio composto da un vocabolario e regole per la formazione delle frasi (enunciati).
2. Una semantica: stabilisce una corrispondenza tra gli enunciati e fatti del mondo. Se un agente ha un enunciato X nella sua KB, crede che il fatto corrispondente sia vero nel mondo.
3. Un meccanismo inferenziale: (codificato o meno tramite regole di inferenza come nella logica) che ci consente di inferire nuovi fatti.

A questo punto, qual è la complessità computazionale di sapere se α è conseguenza logica della KB nei vari linguaggi logici? Quali sono gli algoritmi di decisione e le strategie di ottimizzazione? I linguaggi logici, calcolo proposizionale (PROP) e logica dei predicati (FOL), sono adatti per la rappresentazione della conoscenza?

4 Agenti Logici: Calcolo proposizionale

4.1 Sintassi (regole)

La sintassi definisce quali sono le frasi legittime (ben formate) del linguaggio.

- simbolo $\rightarrow P \mid Q \mid R \dots$
- formula $\rightarrow formulaAtomica \mid formulaComplessa$
- formulaAtomica $\rightarrow Ture \mid False \mid simbolo$
- formulaComplessa $\rightarrow \neg formula \mid formula \wedge formula \mid formula \vee formula \mid formula \Rightarrow formula \mid formula \Leftrightarrow formula$

Possiamo omettere le parentesi assumendo questa precedenza tra gli operatori:

$\neg > \wedge > \vee > \Rightarrow, \Leftrightarrow$

4.2 Semantica (significato)

La semantica definisce il significato di un enunciato, cioè se esso è vero o falso rispetto ad una interpretazione. Una interpretazione definisce un valore di verità per tutti i simboli proposizionali. Il significato di una frase è determinato dal significato dei suoi componenti, a partire dai simboli proposizionali.

4.3 Conseguenza logica

Come abbiamo visto precedentemente, il problema è capire quando, data una base di conoscenza KB contenente una rappresentazione dei fatti che si ritengono veri, un certo fatto α è vero di conseguenza. ($KB \models \alpha$, cioè α è conseguenza logica della KB).

Una formula α è conseguenza logica di un insieme di formule KB se e solo se in ogni modello² di KB anche α è vera.

Indicheremo con $M(\alpha)$ i modelli di α , cioè l'insieme delle interpretazioni che rendono α vera, e con $M(KB)$ i modelli dell'insieme di formule in KB otteniamo:

$$KB \models \alpha \Leftrightarrow M(KB) \subseteq M(\alpha)$$

²Modello: un'attribuzione di un significato a tutti gli enunciati (le formule) del linguaggio che le rende tutte vere

4.4 Il mondo del Wumpus

Il mondo del Wumpus è una caverna fatta di stanze connesse tra di loro. Il Wumpus mangia chiunque entri nella stanza in cui si trova. Il Wumpus può essere ucciso dall'agente, che ha solo una freccia a disposizione. In alcune stanze sono presenti dei pozzi: se l'agente entra in una di queste stanze cade nel pozzo e muore. In una delle stanze si trova l'oro e l'obiettivo dell'agente è di trovarlo e tornare a casa sano e salvo. L'agente non conosce l'ambiente, né la sua locazione. Solo all'inizio sa dove si trova (in [1,1]). In questo problema gli ambienti sono generati a caso e alcuni di questi non sono risolubili (ma sappiamo che lo stato (1,1) è safe): l'agente in alcune situazioni deve decidere se rischiare di essere ucciso pur di trovare l'oro e quando conviene tornare a casa a mani vuote.

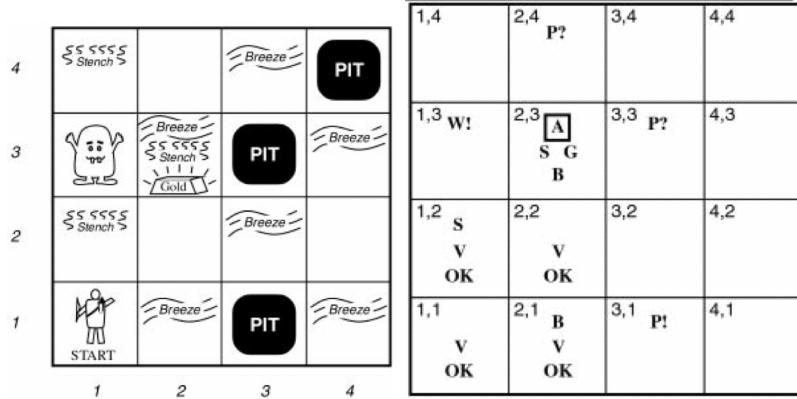


Figure 6: Wumpus' world

- Misura delle prestazioni:
 - +1000 se trova l'oro, torna in [1,1] e esce
 - -1000 se muore
 - -1 per ogni azione
 - -10 se usa la freccia
- Percezioni:
 - **puzzo** nelle caselle adiacenti al Wumpus
 - **brezza** nelle caselle adiacenti alle buche
 - **luccichio** nelle caselle con l'oro
 - **bump** se sbatte in un muro
 - **urlo** se il Wumpus viene ucciso
 - L'agente non percepisce la sua locazione
- Azioni:
 - avanti
 - destra
 - sinistra
 - afferra oggetto
 - scaglia la freccia
 - esci

Le percezioni sono una quintupla [puzzo, brezza, luccichio, bump, urlo]. ([none, none, none, none, none] se non percepisce niente).

4.4.1 Esempio dal mondo del Wumpus

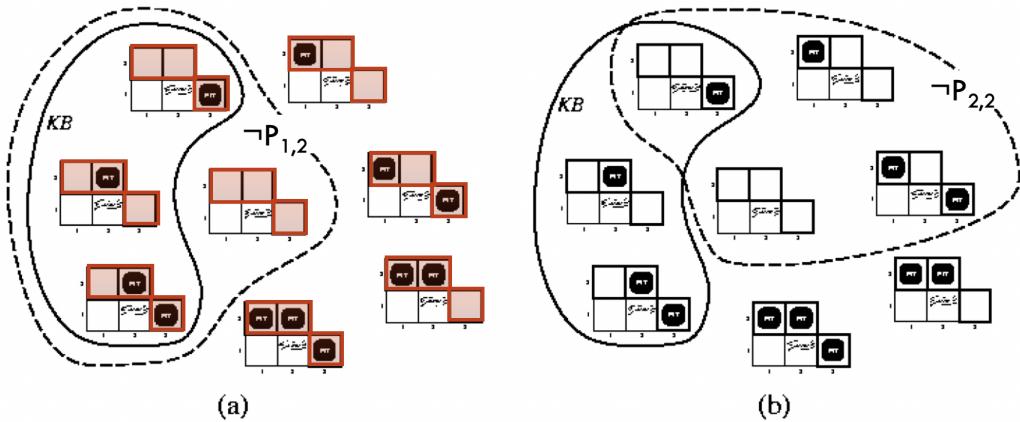
$KB = \{B_{2,1}, \neg B_{1,1}, + \text{le regole del WumpusWorld}\}$

Vogliamo stabilire l'assenza di pozzi in [1,2] e in [2,2]

$KB \models \neg P_{1,2}?$

$KB \models \neg P_{2,2}?$

Ci sono otto possibili interpretazioni o mondi, considerando solo l'esistenza di pozzi nelle 3 caselle $P_{1,2}$, $P_{2,2}$, $P_{3,1}$ (perchè non posso muovermi in diagonale). Guardando le immagini sottostanti, a sinistra prendiamo tutti i mondi in cui non c'è pozzo in [1,2] e notiamo che facendo così includo tutti i mondi della mia KB, quindi $KB \vdash \neg P_{1,2}$. A destra prendo i mondi dove non c'è pozzo in [2,2] e vediamo che questi non includono tutti i mondi della mia KB, quindi $KB \not\vdash \neg P_{2,2}$.



4.5 Formule logiche, Validità e Soddisfacibilità

$$\begin{aligned}
 (\alpha \wedge \beta) &\equiv (\beta \wedge \alpha) \quad \text{cominutativity of } \wedge \\
 (\alpha \vee \beta) &\equiv (\beta \vee \alpha) \quad \text{cominutativity of } \vee \\
 ((\alpha \wedge \beta) \wedge \gamma) &\equiv (\alpha \wedge (\beta \wedge \gamma)) \quad \text{associativity of } \wedge \\
 ((\alpha \vee \beta) \vee \gamma) &\equiv (\alpha \vee (\beta \vee \gamma)) \quad \text{associativity of } \vee \\
 \neg(\neg \alpha) &\equiv \alpha \quad \text{double-negation elimination} \\
 (\alpha \Rightarrow \beta) &\equiv (\neg \beta \Rightarrow \neg \alpha) \quad \text{contraposition} \\
 (\alpha \Rightarrow \beta) &\equiv (\neg \alpha \vee \beta) \quad \text{implication elimination} \\
 (\alpha \Leftrightarrow \beta) &\equiv ((\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)) \quad \text{biconditional elimination} \\
 \neg(\alpha \wedge \beta) &\equiv (\neg \alpha \vee \neg \beta) \quad \text{de Morgan} \\
 \neg(\alpha \vee \beta) &\equiv (\neg \alpha \wedge \neg \beta) \quad \text{de Morgan} \\
 (\alpha \wedge (\beta \vee \gamma)) &\equiv ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma)) \quad \text{distributivity of } \wedge \text{ over } \vee \\
 (\alpha \vee (\beta \wedge \gamma)) &\equiv ((\alpha \vee \beta) \wedge (\alpha \vee \gamma)) \quad \text{distributivity of } \vee \text{ over } \wedge
 \end{aligned}$$

Figure 7: Equivalenze Logiche (leggi)

A valida sse è vera in tutte le interpretazioni (detta anche tautologia).

A soddisfacibile sse esiste una interpretazione in cui A è vera.

A è valida sse $\neg A$ è insoddisfacibile.

4.6 Inferenza per calcolo proposizionale (PROP)

- Model checking: una forma di inferenza che fa riferimento alla definizione di conseguenza logica (si enumerano i possibili modelli e si controllano tramite le tabelle di verità)
- Algoritmi per la soddisfabilità: problemi riconducibili ad altri problemi.
 $KB \models A$ sse $KB \wedge \neg A$ è insoddisfacibile.

4.7 Algoritmo TT-Entails (Model Checking)

Come facciamo a sapere se $KB \models \alpha$? L'algoritmo TT-Entails enumera tutte le possibili interpretazioni di KB (k simboli, 2^k possibili interpretazioni). Per ciascuna interpretazione:

- Se non soddisfa $KB \rightarrow$ OK (non ci interessa, andiamo avanti)
- Se soddisfa $KB \rightarrow$ si controlla che soddisfi anche α .
 Se si trova anche solo una interpretazione che soddisfa KB e non α la risposta è NO.

4.7.1 Esempio TT-Entails

$(\neg A \vee B) \wedge (A \vee C) \vDash (B \vee C) ?$ <ul style="list-style-type: none"> ▪ TT-CHECK-ALL($(\neg A \vee B) \wedge (A \vee C), (B \vee C), [A, B, C], \{ \}$) ▪ TT-CHECK-ALL($(\neg A \vee B) \wedge (A \vee C), (B \vee C), [B, C], \{A=t\}$) <ul style="list-style-type: none"> ▪ TT-CHECK-ALL($(\neg A \vee B) \wedge (A \vee C), (B \vee C), [C], \{A=t, B=t\}$) <ul style="list-style-type: none"> ▪ TT-CHECK-ALL($(\neg A \vee B) \wedge (A \vee C), (B \vee C), [], \{A=t, B=t, C=t\}$) OK ▪ TT-CHECK-ALL($(\neg A \vee B) \wedge (A \vee C), (B \vee C), [], \{A=t, B=t, C=f\}$) OK ▪ TT-CHECK-ALL($(\neg A \vee B) \wedge (A \vee C), (B \vee C), [C], \{A=t, B=f\}$) <ul style="list-style-type: none"> ▪ TT-CHECK-ALL($(\neg A \vee B) \wedge (A \vee C), (B \vee C), [], \{A=t, B=f, C=t\}$) OK ▪ TT-CHECK-ALL($(\neg A \vee B) \wedge (A \vee C), (B \vee C), [], \{A=t, B=f, C=f\}$) OK ▪ TT-CHECK-ALL($(\neg A \vee B) \wedge (A \vee C), (B \vee C), [B, C], [A=f]$) ... 	$Symbols = [A, B, C]$
---	-----------------------

Solo alla fine, dopo avere provato tutti i possibili assegnamenti, possiamo rispondere se Vero o Falso.

4.8 Algoritmo DPLL (Alg. Soddisfabilità)

4.8.1 Forma a clausole

Gli algoritmi per la soddisfabilità usano KB in forma a clausole.

Ad esempio $\{A, B\} \{\neg B, C, D\} \{\neg A, F\}$ è una forma normale congiuntiva cioè una congiunzione di disgiunzioni letterali. Il suo significato quindi è:

$(A \vee B) \wedge (\neg B \vee C \vee D) \wedge (\neg A \vee F)$, la cosa interessante è che non è restrittiva: è sempre possibile ottenerla con trasformazioni che preservano l'equivalenza logica.

I passi da seguire per ottenere la forma a clausole sono:

1. Eliminazione della \Leftrightarrow : $(A \Leftrightarrow B) \equiv (A \Rightarrow B) \wedge (B \Rightarrow A)$
2. Eliminazione dell' \Rightarrow : $(A \Rightarrow B) \equiv (\neg A \vee B)$
3. Negazioni all'interno:
 - $\neg(A \vee B) \equiv (\neg A \wedge \neg B)$ (de Morgan)
 - $\neg(A \wedge B) \equiv (\neg A \vee \neg B)$
4. Distribuzione di \vee su \wedge : $(A \vee (B \wedge C)) \equiv (A \vee B) \wedge (A \vee C)$

L'algoritmo DPLL sfrutta la soddisficiabilità di $KB \models A$ se $KB \wedge \neg A = \{\}$. Parte da una KB in forma a clausole, avviene una enumerazione in profondità di tutte le possibili interpretazioni alla ricerca di un modello. Tre miglioramenti rispetto a TT-Entails:

1. Terminazione anticipata: Si può decidere sulla verità di una clausola anche con interpretazioni parziali, basta che un letterale sia vero. Ad esempio se A è vero lo sono anche $\{A, B\}$ e $\{A, C\}$ indipendentemente dai valori di B e C . (ricordiamo che sono in $\vee!$). Se anche una sola clausola è falsa l'interpretazione non può essere un modello dell'insieme di clausole.
2. Simboli puri: un simbolo puro è un simbolo che appare con lo stesso segno in tutte le clausole. Ad esempio $\{A, \neg B\} \{\neg B, \neg C\} \{C, A\}$ dove A e B sono puri. Nel determinare se un simbolo è puro se ne possono trascurare le occorrenze in clausole già rese vere cioè i simboli puri possono essere assegnati a True se il letterale è positivo, False se negativo. Facendo così non si eliminano modelli utili, se le clausole hanno un modello continuano ad averlo dopo questo assegnamento. L'assegnamento è obbligato.
3. Clausole unitarie: una clausola con un solo letterale non assegnato, ad esempio $\{B\}$ è unitaria ma anche $\{B, \neg C\}$ è unitaria quando $C = \text{True}$. Conviene assegnare prima valori ai letterali in clausole unitarie. L'assegnamento è univoco (True se positivo, False se negativo).

4.8.2 Esempio DPLL

$$KB = (\{\neg B_{1,1}, P_{1,2}, P_{2,1}\} \{\neg P_{1,2}, B_{1,1}\} \{\neg P_{2,1}, B_{1,1}\} \{\neg B_{1,1}\}) \models \{\neg P_{1,2}\}?$$

Aggiungiamo $\{P_{1,2}\}$ (perché sarebbe $\neg(\neg P_{1,2})$) e vediamo se l'insieme è insoddisfacibile:

$$\text{DPLL}(\{\neg B_{1,1}, P_{1,2}, P_{2,1}\} \{\neg P_{1,2}, B_{1,1}\} \{\neg P_{2,1}, B_{1,1}\} \{\neg B_{1,1}\} \{P_{1,2}\})$$

Notiamo che $\{P_{1,2}\}$ è unitaria quindi assegnamo $P_{1,2} = \text{True}$.

La prima clausola $\{\neg B_{1,1}, P_{1,2}, P_{2,1}\}$ e $\{P_{1,2}\}$ sono soddisfatte.

La seconda clausola $\{\neg P_{1,2}, B_{1,1}\}$ diventa unitaria, $B_{1,1} = \text{True}$.

A questo punto $\{\neg P_{1,2}, B_{1,1}\}$ e $\{\neg P_{2,1}, B_{1,1}\}$ sono soddisfatte, ma $\{\neg B_{1,1}\}$ no (perché prima abbiamo assegnato $B_{1,1} = \text{True}$, quindi $\neg B_{1,1} = \text{False}$).

FAIL!

Non esistono modelli, quindi possiamo dire che $\neg P_{1,2}$ è conseguenza logica della KB.

4.8.3 Miglioramenti DPLL

DPLL è completo e termina sempre. Alcuni miglioramenti: analisi di componenti (se le variabili possono essere suddivise in sotto-insiemi disgiunti, cioè senza simboli in comune), ordinamento di variabili e valori (scegliere la variabile che compare in più clausole), backtracking intelligente e altre ottimizzazioni...

4.9 Algoritmo WALK-SAT

E' un algoritmo di ricerca locale, l'obiettivo è un assegnamento che soddisfa tutte le clausole (un modello) partendo da un assegnamento casuale. Ad ogni passo si cambia il valore di una proposizione (flip). Gli stati sono valutati contando il numero di clausole non soddisfatte (meno sono meglio è). Ci sono molti minimi locali, per non incapparci serve introdurre perturbazioni casuali, come succedeva con Hill Climbing con riavvio casuale o Simulated Annealing. WALK-SAT è uno degli algoritmi più semplici ed efficaci.

WALK-SAT ad ogni passo:

- Sceglie a caso una clausola non ancora soddisfatta
- Sceglie un simbolo da modificare (flip) scegliendo con probabilità p (di solito 0,5) tra una delle due:
 - Passo casuale: un simbolo a caso
 - Passo di ottimizzazione: sceglie quello che rende più clausole soddisfatte
- Si arrende dopo un certo numero di flip predefinito (variabile max-flips)

4.9.1 Esempio WALK-SAT

Come euristica usiamo il numero di clausole soddisfatte (valore da massimizzare).

Rosso: passo casuale

Blu: passo di ottimizzazione

(1){ $\neg B_{1,1}, P_{1,2}, P_{2,1}$ } (2){ $\neg P_{1,2}, B_{1,1}$ } (3){ $\neg P_{2,1}, B_{1,1}$ } (4){ $\neg B_{1,1}$ }
 $[B_{1,1} = F, P_{1,2} = T, P_{2,1} = T]$ clausola 2, 3 F; scelgo clausola 2; a caso eseguo flip $B_{1,1}$
 $[B_{1,1} = T, P_{1,2} = T, P_{2,1} = T]$ clausola 4 F; scelgo 4; ottimizzazione: flip $B_{1,1}$ (unica scelta)
 $[B_{1,1} = F, P_{1,2} = T, P_{2,1} = T]$ clausole 2, 3 F; scelgo clausola 2; a caso eseguo flip $P_{1,2}$
 $[B_{1,1} = F, P_{1,2} = F, P_{2,1} = T]$ clausola 3 F; scelgo 3; ottimizzazione: flip $P_{2,1}$
 $[B_{1,1} = F, P_{1,2} = F, P_{2,1} = F]$ modello

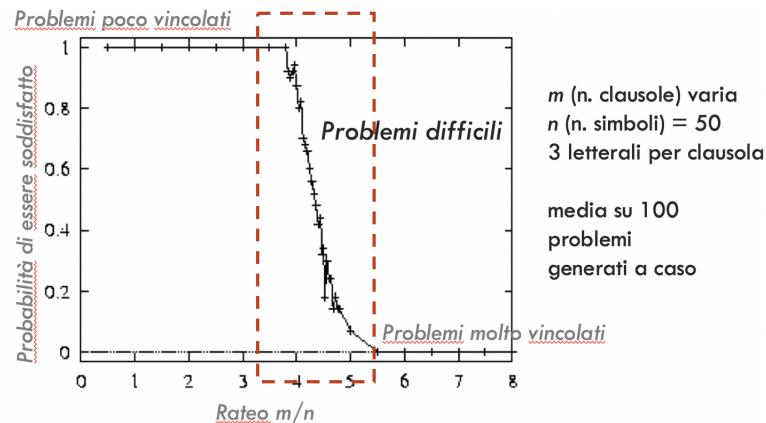
4.9.2 Analisi WALK-SAT

Se $\text{max-flips} = \infty$ e l'insieme di clausole è soddisfacibile prima o poi termina. Va bene per cercare un modello, sapendo che esiste, ma se è insoddisfacibile non termina. Non può essere usato per verificare l'insoddisfacibilità, il problema quindi è decidibile ma l'algoritmo non è completo.

4.9.3 Problemi SAT difficili

Se un problema ha molte soluzioni (problema sotto-vincolato) è più probabile che WALK-SAT ne trovi una in tempi brevi. Ma dato che SAT è un problema NP-Completo, alcune istanze richiedono tempo esponenziale per la risoluzione. Un esempio è che nella seguente istanza esistono 16 soluzioni su 32 assegnamenti possibili, un assegnamento ha il 50% di probabilità di essere giusto: in circa 2 passi random si indovina.

$(\neg D \vee \neg B \vee C) \wedge (B \vee \neg A \vee \neg C) \wedge (\neg C \vee \neg B \vee E) \wedge (E \vee \neg D \vee B) \wedge (B \vee E \vee \neg C)$
quello che conta è il rapporto m/n , dove m è il numero di clausole (vincoli) e n il numero di simboli.
In questo esempio $5/5=1$, più è grande il rapporto, più vincolato è il problema.



4.10 Inferenza come deduzione

Un altro modo per decidere se $\text{KB} \models A$ è usare un meccanismo di deduzione, scriviamo $\text{KB} \vdash A$ (A è deducibile da KB). La deduzione avviene specificando delle regole di inferenza.

- Correttezza: Se $\text{KB} \vdash A$ allora $\text{KB} \models A$. Tutto ciò che è derivabile è conseguenza logica.
- Completezza: Se $\text{KB} \models A$ allora $\text{KB} \vdash A$. Tutto ciò che è conseguenza logica è ottenibile tramite il meccanismo di inferenza.

Nota: questa nozione di completezza (detta anche completezza semantica) si riferisce al teorema di completezza di Gödel. Il teorema di incompletezza di Gödel³ non asserisce che non esistono sistemi deduttivi completi per il FOL, ma che una logica abbastanza potente da assiomatizzare l'aritmetica non è in grado di dimostrare tutte le affermazioni vere sui numeri naturali.

4.10.1 Alcune regole di inferenza

$$\begin{array}{ll} \text{Modus Ponens (eliminazione dell'implicazione)} & \frac{\alpha \Rightarrow \beta \quad \alpha}{\beta} \\ \text{Eliminazione dell'AND} & \frac{\alpha \wedge \beta}{\alpha} \\ \text{Eliminazione doppia implicazione} & \frac{\alpha \Leftrightarrow \beta}{(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)} \\ \text{Introduzione doppia implicazione} & \frac{(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)}{\alpha \Leftrightarrow \beta} \end{array}$$

Come decidere ad ogni passo qual è la regola di inferenza da applicare? E a quali premesse applicarla? Nasce quindi un problema di esplorazione di uno spazio di stati.

Una procedura di dimostrazione definisce:

- Direzione della ricerca: nella dimostrazione di teoremi conviene procedere all'indietro. Se si vuole dimostrare $A \wedge B$ si cerca di dimostrare A e poi B , se si vuole dimostrare $A \Rightarrow B$, si assume A e si cerca di dimostrare B .
- Strategia di ricerca:
 - Completezza: Le regole della deduzione naturale sono un insieme di regole di inferenza completo, se l'algoritmo di ricerca è completo siamo a posto!
 - Efficienza: La complessità è alta, è un problema decidibile ma NP-Completo.

4.11 Regola di risoluzione

Utilizzeremo un'unica regola: la regola di risoluzione (presuppone la forma a clausole).

$$\frac{\{P, Q\} \quad \{\neg P, R\}}{\{Q, R\}} \quad \frac{P \vee Q \quad \neg P \vee R}{Q \vee R}$$

4.11.1 Regola di risoluzione in generale per PROP

$$\frac{\{I_1, I_2, \dots, I_i, \dots, I_k\} \{M_1, M_2, \dots, M_j, \dots, M_n\}}{\{I_1, I_2, \dots, I_{i-1}, I_{i+1}, \dots, I_k, M_1, M_2, \dots, M_{j-1}, M_{j+1}, \dots, M_n\}}$$

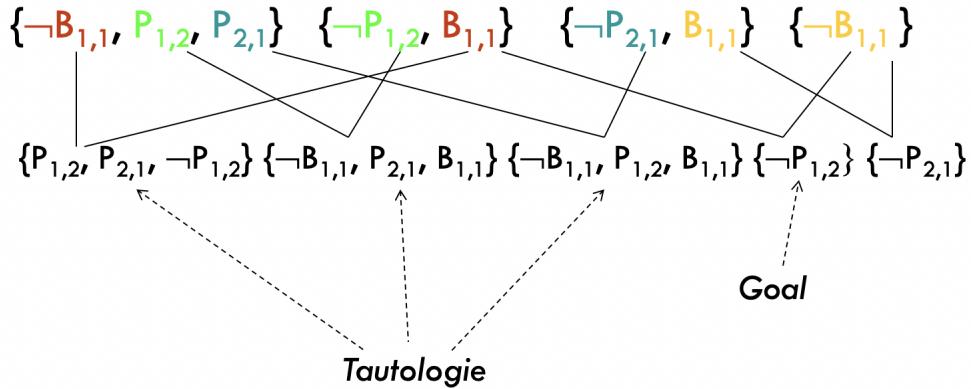
"I" e "M" sono letterali, simboli di proposizione positivi o negativi, la cosa fondamentale da sapere è che I_i e M_j sono simboli uguali ma di segno opposto.

³In ogni formalizzazione coerente della matematica che sia sufficientemente potente da definire la struttura dei numeri naturali dotati delle operazioni di somma e prodotto, è possibile costruire una proposizione sintatticamente corretta che non può essere né dimostrata né confutata all'interno dello stesso sistema.

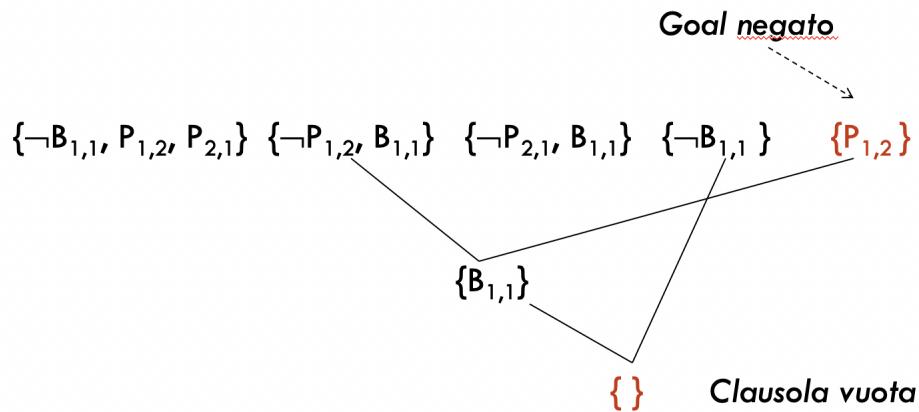
Un caso particolare avviene quando ci troviamo in questa situazione:

$$\frac{\{P\} \quad \{\neg P\}}{\{\}} \text{ clausola vuota} \rightarrow \text{contraddizione}$$

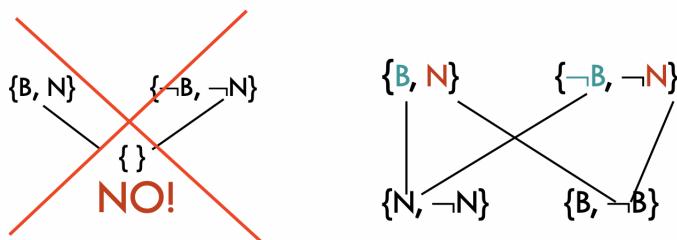
4.11.2 Esempio grafo di risoluzione



4.11.3 Refutazione



ATTENZIONE:



4.11.4 Osservazioni

Abbiamo una singola regola di inferenza, ma è sufficiente? Siamo sicuri che applicando la regola in tutti i modi possibili riesco a dedurre α quando è conseguenza logica? Vale la completezza? Non sempre.

Per fortuna il teorema di refutazione ci offre un modo alternativo di procedere: se voglio sapere se $KB \models \alpha$, aggiungo $\neg\alpha$ a KB e controllo che l'insieme ottenuto sia insoddisfacibile. La correttezza della regola ci dice anche che se da tale insieme derivo la clausola vuota allora in effetti l'insieme è insoddisfacibile. Invece il teorema di risoluzione ci garantisce che se KB è insoddisfacibile allora la clausola vuota sono sempre in grado di trovarla con applicazioni della regola di risoluzione $KB \vdash_{RES} \{\}$. Procedere per refutazione ci garantisce la completezza (naturalmente se la procedura applica la regola in maniera sistematica).

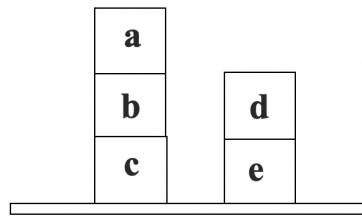
5 Agenti Logici: Logica del prim'ordine

Nella logica dei predicati abbiamo assunzioni che comprendono gli oggetti, le proprietà e le relazioni.

- Gli oggetti: un libro, un evento, una persona... Possono essere identificati relativamente ad altri oggetti o con simboli, oppure mediante funzioni: "la madre di Pietro". L'insieme degli oggetti rilevanti costituiscono il dominio del discorso. (Il dominio potrebbe essere infinito).
- Le proprietà: "la madre di Pietro è simpatica"
- Le relazioni tra gli oggetti: "Pietro è amico di Paolo"

5.1 Il mondo dei blocchi

- Dominio: {a, b, c, d, e} (blocchi)
- Le funzioni: si individuano le funzioni rilevanti che servono per identificare oggetti. Ad esempio la funzione "Hat(x)": dato un blocco "x" identifica il blocco ad esso superiore.
- Le relazioni: si individuano le relazioni interessanti.
 - On = { $\langle a, b \rangle, \langle b, c \rangle, \langle d, e \rangle$ } //a è su b, b è su c, d è su e
 - Clear = {a, d} //non hanno nulla sopra di loro
 - Table = {c, e} //sono poggiati sul tavolo
 - Block = {a, b, c, d, e} //sono blocchi



5.2 Concettualizzazioni

Otteniamo così la concettualizzazione del problema definita come

$\langle \{a, b, c, d, e\}, \{\text{Hat}\}, \{\text{On}, \text{Clear}, \text{Table}, \text{Block}\} \rangle$

Le concettualizzazioni possibili sono infinite, un aspetto importante è il livello di astrazione giusto per gli scopi della rappresentazione. Se fosse rilevante il colore o la grandezza dei blocchi dovremmo introdurre prediciati anche per questi aspetti.

5.3 FOL

5.3.1 Predicati

- Connuttivo $\rightarrow \wedge | \vee | \neg | \Rightarrow | \Leftrightarrow | \Leftarrow$
- Quantificatore $\rightarrow \forall | \exists$
- Variabile $\rightarrow x | y | \dots$
- Costante $\rightarrow A | B | \text{Mario} | 2 \dots$
- Funzione $\rightarrow \text{Hat} | + | - \dots$
- Predicato $\rightarrow \text{On} | \text{Clear} | > | < | \dots$

L'arietà è il numero di "parametri" della funzione o del predicato.

5.3.2 Termini

- Termine \rightarrow Costante | Variabile | Funzione(Termine,...)

5.3.3 Formule

- Formula-Atomica \rightarrow True | False | Termine=Termine | Predicato(Termine,...)
- Formula \rightarrow Formula-Atomica | Formula Connuttivo Formula | Quantificatore Variabile Formula | \neg Formula

Di solito le variabili sono usate nell'ambito di quantificatori. In tal caso le occorrenze si dicono legate. Se non sono legate, si dicono libere.

$\text{Mela}(x) \Rightarrow \text{Rossa}(x)$ x è libera in entrambe le occorrenze

$\forall x \text{ Mela}(x) \Rightarrow \text{Rossa}(x)$ x è legata

$\text{Mela}(x) \Rightarrow \exists x \text{ Rossa}(x)$ la prima x è libera, la seconda legata

Formula chiusa: una formula che non contiene occorrenze di variabili libere (altrimenti è detta aperta).

Formula ground: una formula che non contiene variabili.

5.3.4 Interpretazione

La semantica dichiarativa consiste nello stabilire una corrispondenza tra i termini del linguaggio e gli oggetti del mondo.

Una interpretazione "I" stabilisce una corrispondenza precisa tra elementi atomici del linguaggio ed elementi della concettualizzazione. "I" interpreta:

- i simboli di costante come elementi del dominio
- i simboli di funzione come funzioni da n-uple di $D \rightarrow D$
- i simboli di predicato come insiemi di n-uple

5.4 Sematica Composizionale

Il significato di un termine o di una formula composta è determinato in funzione del significato dei suoi componenti.

- Semantica \forall : $\forall x A(x)$ è vera se per ciascun elemento del dominio A è vera. Se il dominio è finito equivale a un grosso \wedge . Tipicamente, siccome difficilmente una proprietà è universale, \forall si usa quasi sempre insieme a \Rightarrow . (Es: $\forall x \text{ Persona}(x) \Rightarrow \text{Mortale}(x)$).
- Semantica \exists : $\exists x A(x)$ è vera se esiste almeno un elemento del dominio per cui A è vera. Se il dominio è finito equivale a un grosso \vee . Tipicamente \exists si usa con \wedge (Es: $\exists x \text{ Persona}(x) \wedge \text{Speciale}(x)$).

$$\begin{array}{ll}
 \forall x \neg P(x) \equiv \neg \exists x P(x) & \neg P \wedge \neg Q \equiv \neg(P \vee Q) \\
 \neg \forall x P(x) \equiv \exists x \neg P(x) & \neg(P \wedge Q) \equiv \neg P \vee \neg Q \\
 \forall x P(x) \equiv \neg \exists x \neg P(x) & P \wedge Q \equiv \neg(\neg P \vee \neg Q) \\
 \neg \forall x \neg P(x) \equiv \exists x P(x) & P \vee Q \equiv \neg(\neg P \wedge \neg Q)
 \end{array}$$

Figure 8: Relazione tra \forall ed \exists

5.4.1 Semantica Standard VS Semantica Database

- Standard: ci permette di inferire.
- Database: abbiamo simboli distinti, oggetti distinti, tutto ciò di cui non si sa che è vero è falso, infine esistono solo gli oggetti di cui si parla.

5.5 Interazione con la KB tramite FOL

- Asserzioni: vengono aggiunte le informazioni alla KB, esempio $\text{TELL}(\text{KB}, \text{King}(\text{John}))$ oppure $\text{TELL}(\text{KB}, \forall x \text{ King}(x) \Rightarrow \text{Person}(x))$
- Conseguenze logiche: $\text{ASK}(\text{KB}, \text{Person}(\text{John}))$ risponde si, se $\text{KB} \models \text{Person}(\text{John})$. Un altro esempio è $\text{ASK}(\text{KB}, \exists x \text{ Person}(x))$ risponde con una lista di sostituzioni o legami: $\{\{x/\text{John}\} \{x/\text{George}\} \dots\}$

Il metodo di risoluzione per FOL si basa sulla trasformazione in forma a clausole e sull'unificazione. Ci sono anche altri metodi particolari, chiamati sistemi a regole, che vedremo alla fine: il "Backward chaining" insieme alla programmazione logica e il "Forward chaining" con basi di dati deduttive.

5.6 Regola di inferenza per \forall

Istanziamento dell'Universale (eliminazione del \forall)
$$\frac{\forall x A[x]}{A[g]}$$
 dove g è un termine ground (ovvero un termine che non contiene variabili) e $A[g]$ è il risultato della sostituzione di g al posto di x in A .
Esempio: $\forall x \text{King}(x) \wedge \text{Greedy}(x) \Rightarrow \text{Evil}(x)$ si ottiene $\text{King}(\text{John}) \wedge \text{Greedy}(\text{John}) \Rightarrow \text{Evil}(\text{John})$

5.7 Regola di inferenza per \exists

Istanziamento dell'esistenziale (eliminazione del \exists)
$$\frac{\exists x A[x]}{A[k]}$$

dove \exists non compare nell'ambito di \forall e k è una costante nuova (costante di Skolem)
Esempio: $\exists x Padre(x, G)$ diventa $Padre(k, G)$.

Nel caso di variabili quantificate universalmente va introdotta una funzione (di Skolem)
Esempio: $\forall x \exists y Padre(x, y)$ diventa $\forall x Padre(x, p(x))$

5.8 Riduzione a inferenza proposizionale

Per semplificare il simbolo di quantificazione potremmo sfruttare un meccanismo di proposizionalizzazione, cioè creare tante istanze delle formule quantificate universalmente quanti sono gli oggetti menzionati ed eliminare i quantificatori esistenziali "skolemizzando". A questo punto possiamo trattare la KB come proposizionale e applicare gli algoritmi visti. Però c'è un ovvio problema: le costanti sono in numero finito ma se ci sono funzioni il cui numero di istanze da creare è infinito, come si procede? In aiuto viene il Teorema di Herbrand.

5.8.1 Teorema di Herbrand

Se $KB \models A$ allora c'è una dimostrazione che coinvolge solo un sotto-insieme finito della KB proposizionalizzata. Si può procedere in modo incrementale, iniziando col creare le istanze con le costanti, creare poi quelle con un solo livello di annidamento $Padre(John)$, $Madre(John)$ ed infine quelle con due livelli di annidamento $Padre(Padre(John))$, $Padre(Madre(John))$, ecc... Se $KB \not\models A$ il processo non termina (problema semidecidibile).

5.9 Verso un metodo di risoluzione per il FOL

Per trovare un metodo di risoluzione, come abbiamo fatto per il PROP, dobbiamo estendere al FOL la trasformazione in forma a clausole e dobbiamo introdurre il concetto di unificazione.

5.9.1 Forma a clausole

Una clausola è un insieme di letterali, che rappresenta la loro disgiunzione.

- Clausola $\rightarrow \{\text{Letterale}, \dots, \text{Letterale}\}$
- Letterale $\rightarrow \text{Formula-Atomica} \mid \neg \text{Formula-Atomica}$

Una KB è un insieme di clausole.

Teorema: per ogni formula chiusa α del FOL è possibile trovare in maniera effettiva un insieme di clausole $Forma_a_Clausole(\alpha)$ che è soddisfacibile se solo se α lo era (o insoddisfacibile se solo se α lo era).

5.9.2 Esempio di trasformazione (passo passo)

Esempio di trasformazione in dettaglio per la frase:

“Tutti coloro che amano tutti gli animali sono amati da qualcuno”

La nostra formula di partenza è: $\forall x(\forall y \text{Animale}(y) \Rightarrow \text{Ama}(x, y)) \Rightarrow (\exists y \text{Ama}(y, x))$

1. Eliminazione delle implicazioni ($\Rightarrow, \Leftrightarrow$):

- $A \Rightarrow B$ diventa $\neg A \vee B$
- $A \Leftrightarrow B$ diventa $(\neg A \vee B) \wedge (\neg B \vee A)$

$$\begin{aligned} \forall x(\forall y \text{Animale}(y) \Rightarrow \text{Ama}(x, y)) &\Rightarrow (\exists y \text{Ama}(y, x)) \\ \forall x(\forall y \text{Animale}(y) \Rightarrow \text{Ama}(x, y)) &\vee (\exists y \text{Ama}(y, x)) \\ \forall x(\forall y \neg \text{Animale}(y) \vee \text{Ama}(x, y)) &\vee (\exists y \text{Ama}(y, x)) \end{aligned}$$

2. Negazioni all'interno:

- $\neg\neg A$ diventa A
- $\neg(A \wedge B)$ diventa $\neg A \vee \neg B$
- $\neg(A \vee B)$ diventa $\neg A \wedge \neg B$
- $\neg\forall x A$ diventa $\exists x \neg A$
- $\neg\exists x A$ diventa $\forall x \neg A$

$$\begin{aligned} \forall x \neg(\forall y \neg \text{Animale}(y) \vee \text{Ama}(x, y)) &\vee (\exists y \text{Ama}(y, x)) \\ \forall x(\exists y \neg(\neg \text{Animale}(y) \vee \text{Ama}(x, y))) &\vee (\exists y \text{Ama}(y, x)) \\ \forall x(\exists y \text{Animale}(y) \wedge \neg \text{Ama}(x, y)) &\vee (\exists y \text{Ama}(y, x)) \end{aligned}$$

3. Standardizzazione delle variabili: ogni quantificatore una variabile diversa

$$\forall x(\exists y \text{Animale}(y) \wedge \neg \text{Ama}(x, y)) \vee (\exists z \text{Ama}(z, x))$$

4. Skolemizzazione: eliminazione dei quantificatori esistenziali.

In questo caso essendo che i due quantificatori esistenziali sono nell'ambito di uno universale dobbiamo introdurre due funzioni di Skolem.

$$\forall x(\text{Animale}(F(x)) \wedge \neg \text{Ama}(x, F(x))) \vee (\text{Ama}(G(x), x))$$

5. Eliminazione quantificatori universali: possiamo portarli tutti davanti e poi eliminarli usando la convenzione che le variabili libere sono quantificate universalmente.

$$(\text{Animale}(F(x)) \wedge \neg \text{Ama}(x, F(x))) \vee (\text{Ama}(G(x), x))$$

6. Forma normale congiuntiva (congiunzione di disgiunzioni di letterali):

$$(\text{Animale}(F(x)) \vee (\text{Ama}(G(x), x))) \wedge (\neg \text{Ama}(x, F(x)) \vee (\text{Ama}(G(x), x)))$$

7. Notazione a clausole

$$\{\text{Animale}(F(x)), (\text{Ama}(G(x), x))\} \{\neg \text{Ama}(x, F(x)), (\text{Ama}(G(x), x))\}$$

8. Separazione delle variabili: clausole diverse, variabili diverse

$$\{\text{Animale}(F(x_1)), (\text{Ama}(G(x_1), x_1))\} \{\neg \text{Ama}(x_2, F(x_2)), (\text{Ama}(G(x_2), x_2))\}$$

5.10 Sostituzione

Possiamo eseguire la sostituzione in un insieme finito di associazioni tra variabili e termini, in cui ogni variabile compare una sola volta sulla sinistra. Ad esempio $\{x_1/A, x_2/f(x_3), x_3/B\}$, significa che x_1 va sostituita con A , x_2 va sostituito con $f(x_3)$ e x_3 con B (nota: sulla sinistra sono solo variabili!)

Sia σ una sostituzione e A un'espressione: $A\sigma$ è l'istanza generata dalla sostituzione (delle variabili con le corrispondenti espressioni). Possiamo scriverlo anche come $\text{Subst}(\sigma, A)$.

Esempio:

$$\text{Subst}(\{x/A, y/f(B), z/W\}, P(x, x, y, v)) = P(A, A, f(B), v)$$

$$\text{Subst}(\{x/g(y), y/z, z/f(x)\}, Q(x, y, z)) = Q(g(y), z, f(x))$$

5.11 Unificazione

L'unificazione è una operazione che ci permette di determinare se due espressioni possono essere rese identiche mediante una sostituzione di termini a variabili. Il risultato è la sostituzione che rende le due espressioni identiche detta unificatore, oppure restituisce FAIL se le espressioni non sono unificabili.

Esempio: $P(A, y, z)$ e $P(x, B, z)$ sono unificabili con $\tau = \{x/A, y/B, z/C\}$

possiamo notare come τ sia un unificatore, ma non l'unico, un altro è $\sigma = \{x/A, y/B\}$.

Possiamo infine notare come σ è più generale di τ , cioè istanzia "meno" variabili. Noi però vorremmo l'unificatore più generale di tutti (MostGeneralUnifier - MGU) e abbiamo un teorema che dice che l'unificatore più generale è unico, a parte i nomi delle variabili (l'ordine non conta).

5.11.1 Algoritmo di unificazione

L'algoritmo di unificazione prende in input due espressioni p, q e restituisce un MGU Θ se esiste. $\text{UNIFY}(p, q) = \Theta$ tale che $\text{SUBST}(\Theta, p) = \text{SUBST}(\Theta, q)$, altrimenti FAIL. L'algoritmo esplora in parallelo le due espressioni e costruisce l'unificatore strada facendo. Appena trova espressioni non unificabili fallisce. Una causa di fallimento sono sostituzioni del tipo $x=f(x)$, cioè sostituzioni in cui x occorre già all'interno dell'espressione. Questo controllo si chiama Occurr-Check() e in tal caso restituisce FAIL (è un controllo di complessità quadratica).

5.11.2 Esempi

Esempio 1

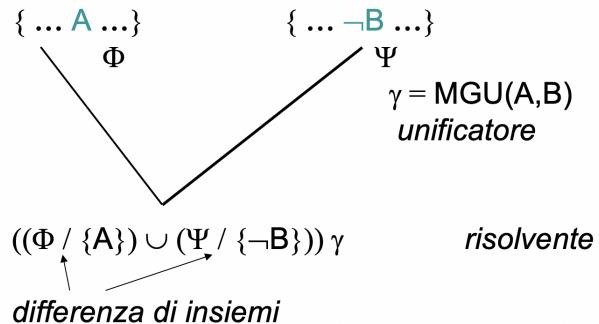
- $\text{UNIFY}(P(A, y, z), P(x, B, z), \{\})$
- $\text{UNIFY}((A, y, z), (x, B, z), \text{UNIFY}(P, P, \{\}))$
- $\text{UNIFY}((A, y, z), (x, B, z), \{ \})$
- $\text{UNIFY}((y, z), (B, z), \text{UNIFY}(A, x, \{\}))$
- $\text{UNIFY}((y, z), (B, z), \text{UNIFY}(x, A, \{\}))$
- $\text{UNIFY}((y, z), (B, z), \text{UNIFY-VAR}(x, A, \{\}))$
- $\text{UNIFY}((y, z), (B, z), \{x/A\})$
- $\text{UNIFY}((z), (z), \{y/B, x/A\})$
- $\text{UNIFY}(z, z, \{y/B, x/A\})$
- $\{y/B, x/A\}$

Esempio 2

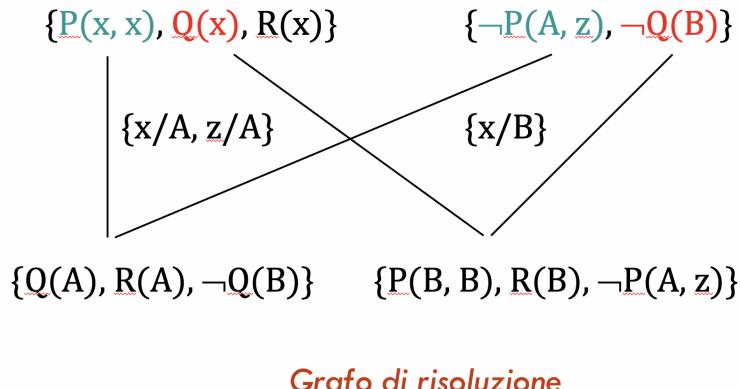
- UNIFY($P(f(x), x)$, $P(z, z)$, {})
- UNIFY(($f(x), x$), (z, z), UNIFY(P, P , {}))
- UNIFY(($f(x), x$), (z, z), {})
- UNIFY((x, z), UNIFY($f(x), z$, {}))
- UNIFY((x, z), UNIFY($z, f(x)$, {}))
- UNIFY((x, z), $\{z/f(x)\}$)
- UNIFY-VAR($x, z, \{z/f(x)\}$)
- UNIFY($x, f(x), \{z/f(x)\}$)
- OCCUR-CHECK($x, f(x)$)
- FAIL

5.12 Metodo di risoluzione per FOL

Siamo ora in grado di definire in generale la regola di risoluzione per FOL.

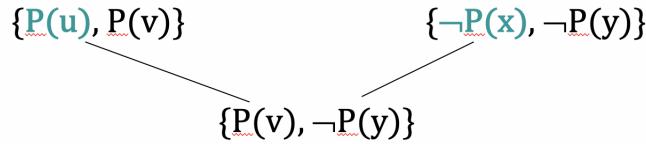


5.12.1 Esempio metodo di risoluzione



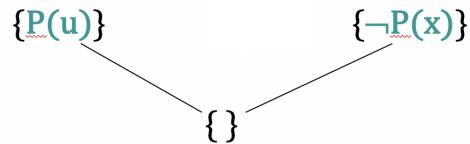
5.12.2 Problemi

Le seguenti clausole dovrebbero produrre la clausola vuota invece...



Se un sottoinsieme dei letterali di una clausola può essere unificato allora la clausola ottenuta dopo tale unificazione si dice fattore della clausola originaria.

Il metodo di risoluzione va applicato ai fattori delle clausole:



5.12.3 Completezza del metodo di risoluzione

La deduzione per risoluzione è corretta se $\Gamma \vdash_{RES} A$ allora $\Gamma \models A$

La deduzione per risoluzione non è completa perché può essere che $\Gamma \models A$ ma non $\Gamma \vdash_{RES} A$

Un esempio è $\{\} \models \{P, \neg P\}$ ma non è vero che $\{\} \vdash_{RES} \{P, \neg P\}$

5.13 Refutazione

Come per PROP, il teorema di refutazione ci suggerisce un metodo alternativo completo. Il teorema di refutazione dice che $\Gamma \cup \{\neg A\}$ è insoddisfacibile sse $\Gamma \models A$.

Possiamo dire quindi che Γ è insoddisfacibile sse $\Gamma \vdash_{RES} \{\}$.

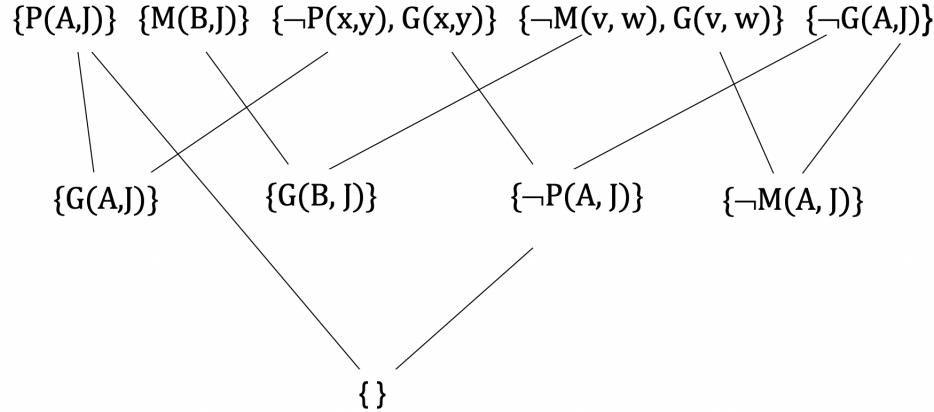
Abbiamo quindi un metodo meccanizzabile, corretto e completo: basta aggiungere il negato della formula da dimostrare e provare a generare la clausola vuota.

5.13.1 Esempio di Refutazione

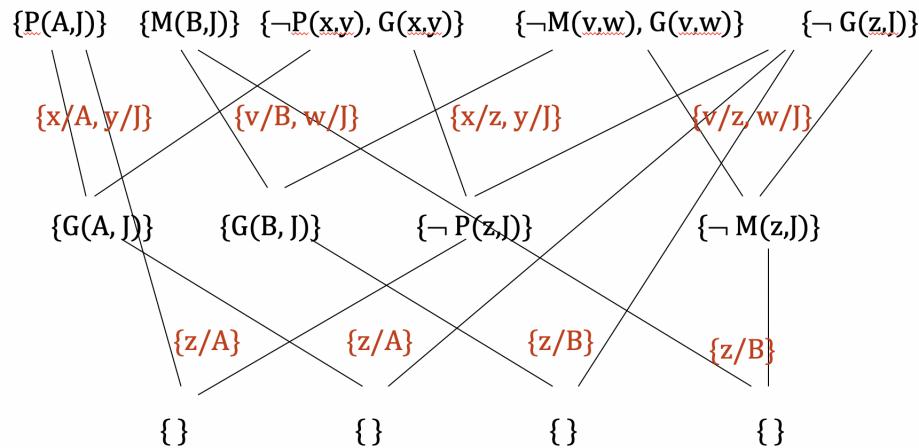
La nostra KB è formata da:

- $\{P(A, J)\}$ A è padre di J
- $\{M(B, J)\}$ B è madre di J
- $\{\neg P(x, y), G(x, y)\}$ padre implica genitore
- $\{\neg M(v, w), G(v, w)\}$ madre implica genitore

Il nostro goal è sapere se A è genitore di J, in forma a clausole se $\{(A, J)\}$ appartiene alla KB. Per refutazione possiamo quindi aggiungere alla KB la negazione del goal $\{\neg G(A, J)\}$ e proviamo a dedurre la clausola vuota.



E per domande del tipo "chi sono i genitori di J?"
Si cerca di dimostrare che $\exists z G(z, J) \rightarrow \{\neg G(z, J)\}$ e la risposta sono tutti i possibili legami per z che consentono di ottenere la clausola vuota.



Le risposte sono: A, B

5.14 Risoluzione efficiente per FOL

Come si può rendere più efficiente il metodo di risoluzione per FOL? Sfrutteremo tecniche per esplorare in maniera efficiente il grafo di risoluzione, probabilmente senza perdere completezza. Distingueremo tra:

- Strategie di cancellazione: ci sono clausole che posso eliminare?
- Strategie di restrizione: posso usare ad ogni passo solo alcune clausole?
- Strategie di ordinamento: posso 'risolvere' i letterali in un ordine specifico?

5.14.1 Strategie di cancellazione

Si tratta di rimuovere dalla KB (ai fini della dimostrazione) certe clausole che non potranno mai essere utili nel processo di risoluzione.

- Clausole con letterali puri: sono quelle clausole che non hanno il loro negato nella KB, ad esempio in $\{\neg P, \neg Q, R\} \{\neg P, S\} \{\neg Q, S\} \{P\} \{Q\} \{\neg R\}$ la seconda e terza clausola contengono il letterale puro S. Le clausole con letterali puri non potranno mai essere risolte con altre clausole per ottenere $\{\}$, tanto vale eliminarle che non si perdono soluzioni.

- Tautologie: la tautologie vengono rappresentate da quelle clausole che contengono due letterali identici e complementari. Ad esempio: $\{P(A), \neg P(A), \dots\}$ oppure $\{P(x), Q(y), \neg Q(y)\}$. La loro rimozione non influenza la soddisfacibilità.

NOTA: Le tautologie possono essere generate, quindi è un controllo da fare ad ogni passo.

- Clausole sussunte⁴: dovremo eliminare le clausole implicate. In generale: α sussume β se solo se $\exists \sigma$ tale che $\alpha\sigma \subseteq \beta$ cioè se un'istanza di α con la sostituzione σ è un sottoinsieme di β .

Ad esempio: $\{P(x), Q(y)\}$ sussume $\{P(A), Q(v), R(w)\}$ infatti $\{P(x), Q(y)\} \{x/A, y/v\} = \{P(A), Q(v)\} \subseteq \{P(A), Q(v), R(w)\}$.

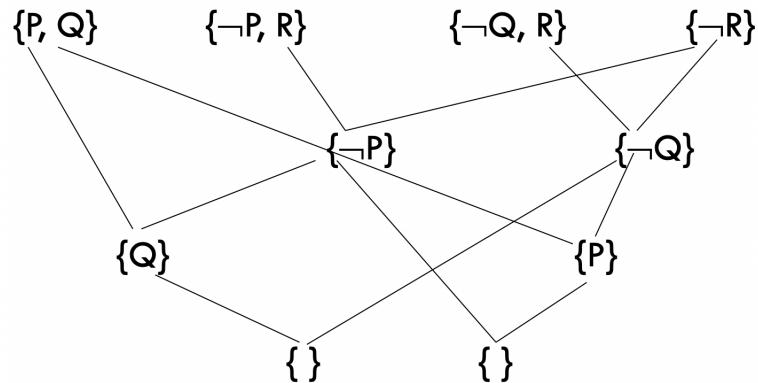
Concludendo, β può essere ricavata da α , quindi β può essere eliminata senza perdere soluzioni.

NOTA: Le clausole sussunte possono essere generate, quindi è un altro controllo da fare ad ogni passo.

5.14.2 Strategie di restrizione

Ad ogni passo si sceglie tra un sottoinsieme delle possibili clausole e si applica l'algoritmo su di esse. Tra le strategie di restrizione possibili:

- Risoluzione unitaria: almeno una delle clausole è unitaria (contiene un solo letterale)



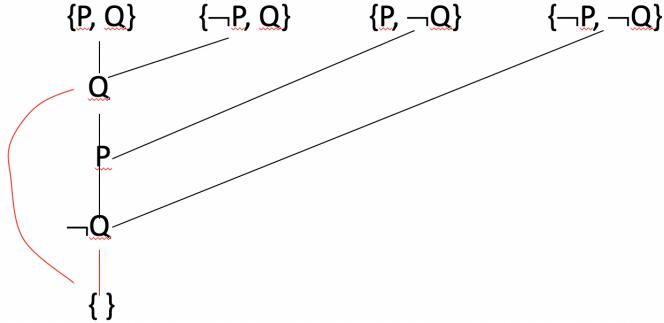
Facile da implementare, converge rapidamente ma la strategia non è completa.

Esempio: $\{P, Q\} \{\neg P, Q\} \{P, \neg Q\} \{\neg P, \neg Q\} \vdash_{RES} \{\}$, ma non otteniamo lo stesso risultato con la risoluzione a clausole unitarie.

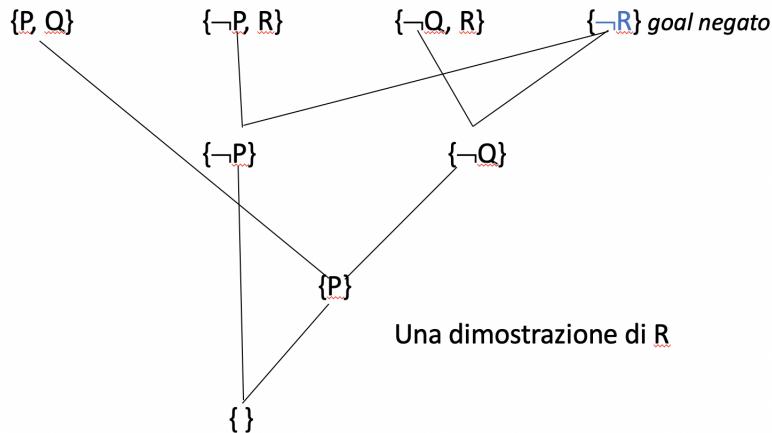
La strategia però è completa per clausole Horn, cioè quelle clausole con al massimo un letterale positivo (ma possono avere più letterali nella stessa clausola), nell'esempio di prima $\{P, Q\}$ non è una clausola Horn!

⁴sussume significa “è più generale di” o “implica”

- Risoluzione lineare: sfruttiamo l'ultima clausola generata con una clausola da input (dalla KB iniziale) oppure una clausola antenata. La risoluzione è completa se applicata insieme ad una refutazione.

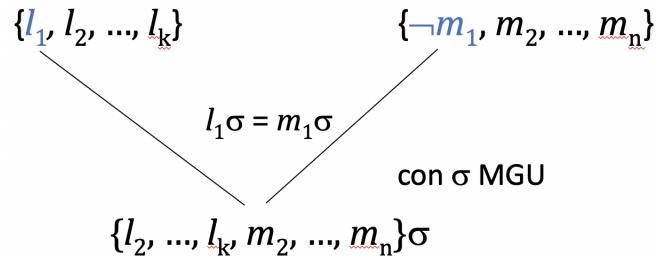


- Risoluzione guidata dal goal: sfruttiamo un insieme di supporto, cioè un sotto-insieme della KB responsabile dell'insoddisfacibilità. Tipicamente, assumendo la KB iniziale consistente, si sceglie come insieme di supporto iniziale il negato della clausola goal (è come procedere all'indietro dal goal). Come si può notare, la strategia è completa per la refutazione.



5.14.3 Strategie di ordinamento

Ogni clausola è un insieme ordinato di letterali e si possono unificare solo i primi letterali delle clausole, l'ordinamento deve essere rispettato nel risolvente. La risoluzione ordinata è completa per clausole Horn.



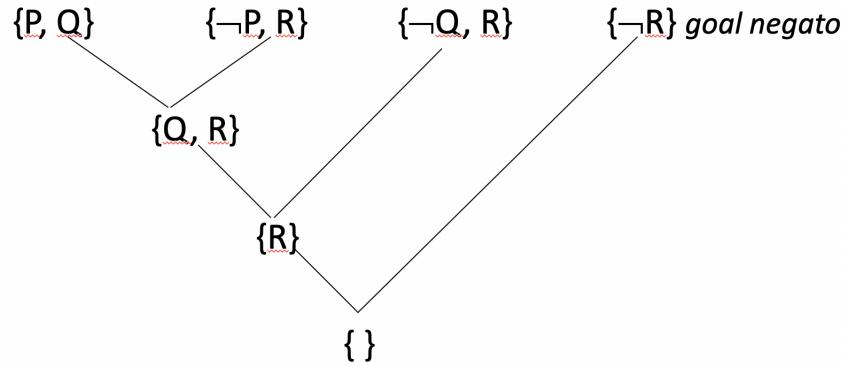


Figure 9: risoluzione ordinata esempio

5.15 Sistemi a regole

Le clausole Horn definite sono quelle clausole con esattamente un letterale positivo⁵. Possono essere riscritte come fatti e regole, cioè

$\neg P_1 \vee \dots \vee \neg P_k \vee Q$ viene riscritta come

$\neg(P_1 \wedge \dots \wedge P_k) \vee Q$ quindi

$(P_1 \wedge \dots \wedge P_k) \Rightarrow Q$ con Q chiamato fatto.

Se la KB contiene solo clausole Horn definite, i meccanismi inferenziali sono molto più semplici senza rinunciare alla completezza. Ovviamente è un sistema restrittivo, non può coincidere con FOL.

5.15.1 Regole in avanti e indietro

- Concatenazione all'indietro (Backward Chaining): approccio guidato dall'obiettivo. Le regole sono applicate alla rovescia. (Programmazione Logica - PROLOG)
- Concatenazione in avanti (Forward Chaining): approccio guidato dai dati. Le regole sono applicate nel senso "antecedente-conseguente" (Basi di dati deduttive)

5.16 Programmazione logica (Backward Chaining)

I programmi logici sono KB costituiti da clausole Horn definite, vengono espressi come fatti e regole, con una sintassi alternativa:

A fatto
 A :- B_1, B_2, \dots, B_n regola, con testa (la conseguenza) A

Come convenzione, in programmazione logica, le variabili sono indicate con lettere maiuscole mentre le costanti con lettere minuscole.

Esempio: Se

$B_1 \wedge B_2 \wedge \dots \wedge B_n$ è il goal,
 $\neg(B_1 \wedge B_2 \wedge \dots \wedge B_n) \vee False$ è il goal negato, ovvero
 $(B_1 \wedge B_2 \wedge \dots \wedge B_n) \Rightarrow False$ che viene scritto
 $\neg B_1, \neg B_2, \dots, \neg B_n$ omettendo il conseguente.

⁵Clausole Horn: possono avere più letterali ma al massimo uno solo positivo
 Clausole Horn Definite: possono avere più letterali ma hanno esattamente un letterale positivo

Nei programmi logici abbiamo due tipi di interpretazione per le regole:

1. Interpretazione dichiarativa: $A :- B_1, B_2, \dots, B_n$ quindi A è vero se sono veri B_1, B_2, \dots, B_n (in accordo al significato logico dell'implicazione).
2. Interpretazione procedurale: la testa può essere vista come una chiamata di procedura e il corpo come una serie di procedure da eseguire in sequenza.

5.16.1 Risoluzione SLD

La risoluzione SLD (Selection Linear Definite-clauses) è una strategia ordinata, basata su un insieme di supporto formato dalla clausola goal ed è lineare in input. La risoluzione SLD è completa per clausole Horn.

5.16.2 Alberi di risoluzione SLD

Dato un programma logico P, l'albero SLD per un goal G è definito come segue:

- ogni foglia dell'albero corrisponde a un goal
- la radice è $:- G_1, G_2, \dots, G_k$ che è il nostro goal
- sia $:- G_1, G_2, \dots, G_k$ un nodo dell'albero; il nodo ha tanti discendenti quanti sono i fatti e le regole in P la cui testa è unificabile con G_1 .
Se $A :- B_1, \dots, B_k$ e A è unificabile con G_1 e $\gamma = \text{MGU}(A, G_1)$
un discendente è il goal $:(-B_1, \dots, B_k, G_2, \dots, G_k)\gamma$
- i nodi che sono clausole vuote sono successi
- i nodi che non hanno successori sono fallimenti

5.16.3 Esempio di albero SLD

Abbiamo le seguenti regole:

1. Genitore(X, Y) :- Padre(X, Y).
2. Genitore(X, Y) :- Madre(X, Y).
3. Antenato(X, Y) :- Genitore(X, Y).
4. Antenato(X, Y) :- Genitore(X, Z), Antenato(Z, Y).
5. Padre(gio, mark).
6. Padre(gio, luc).
7. Madre(lia, gio).
8. $:- \text{Antenato}(\text{lia}, \text{mark})$. (goal negato, quindi radice del nostro albero)

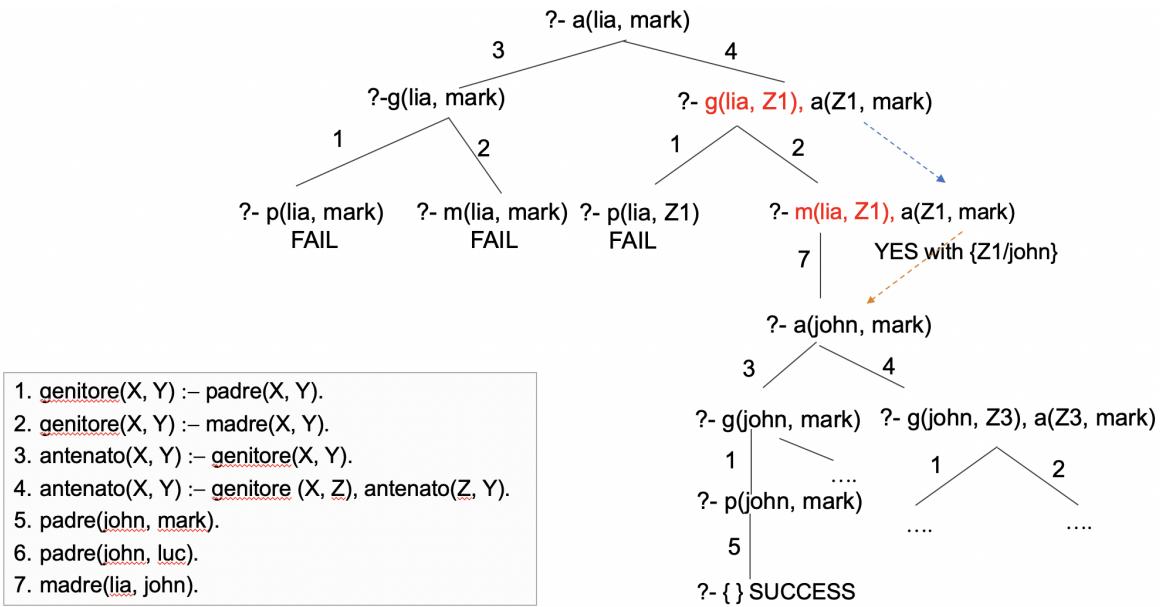


Figure 10: Albero SLD per il goal `antenato(lia, mark)`

La strategia è completa per clausole Horn definite. Se $P \cup \{\neg G\}$ è insoddisfacibile, allora una delle foglie deve essere la clausola vuota (successo).

A seconda di come l'albero viene visitato si potrebbe anche non trovare la clausola vuota, poiché potremmo trovare un cammino infinito e non trovare mai la soluzione. La strategia di ricerca può essere responsabile dell'incompletezza! In PROLOG, il più famoso linguaggio di programmazione logica, la visita dell'albero di risoluzione avviene con una ricerca in profondità, con backtracking in caso di fallimento. La strategia di PROLOG non è completa, le regole vengono applicate nell'ordine in cui sono immesse, infine PROLOG omette l'Occur-Check per motivi di efficienza.

5.17 Sistemi a regole in avanti (Forward Chaining - FOL_FC_Ask)

$$\frac{p'_1, p'_2, \dots, p'_n \quad (p_1 \wedge p_2 \wedge \dots \wedge p_n \Rightarrow q)}{(q)\Theta}$$

Conosciamo la regola del Modus Ponens generalizzato:

con $\Theta = \text{MGU}(p'_i, p_i)$, per ogni i

ma in realtà per applicare la regola dovremo: istanziare gli universali, istanziare le regole e poi applicare il Modus Ponens classico.

Supponiamo di avere nella KB: `King(John)`, `Greedy(y)` e `King(x) \wedge Greedy(x) \Rightarrow Evil(x)`.

Con $\Theta = \{x/John, y/John\}$ si ottiene:

`King(John), Greedy(John) e King(John) \wedge Greedy(John) \Rightarrow Evil(John)`.

Quindi la conclusione della regola è `Evil(John)`.

5.17.1 Esempio di concatenazione in avanti

È un crimine per un Americano vendere armi a una nazione ostile. Il paese Nono, un nemico dell'America, ha dei missili, e tutti i missili gli sono stati venduti dal colonnello West, un Americano.

Dimostrare che West è un criminale. Formalizzazione:

- $American(x) \wedge Arma(y) \wedge Vende(x, y, z) \wedge Ostile(z) \Rightarrow Criminale(x)$
- $\exists x Possiede(Nono, x) \wedge Missile(x)$ lo istanziamo con $Possiede(Nono, M_1) \wedge Missile(M_1)$
- $Missile(x) \wedge Possiede(Nono, x) \Rightarrow Vende(West, x, Nono)$
- $Missile(x) \Rightarrow Arma(x)$
- $Nemico(x, America) \Rightarrow Ostile(x)$
- $American(West)$
- $Nemico(Nono, America)$

Un semplice processo inferenziale chiamato FOL_FC_Ask, applica ripetutamente il Modus Ponens generalizzato per ottenere nuovi fatti fino a che si dimostra quello che si desidera o nessun fatto nuovo può essere aggiunto. E' una strategia di ricerca sistematica in ampiezza, in questo esempio non ci sono funzioni e il processo converge: siamo nelle condizioni di un database DATALOG (cioè basi di dati deduttive). Come si procede:

1. $Possiede(Nono, M_1) \wedge Missile(M_1)$
2. $Missile(x) \wedge Possiede(Nono, x) \Rightarrow Vende(West, x, Nono)$ è soddisfatta con $\{x/M_1\}$ e viene aggiunto $Vende(West, M_1, Nono)$
3. $Missile(x) \Rightarrow Arma(x)$ è soddisfatta con $\{x/M_1\}$ e viene aggiunto $Arma(M_1)$
4. sappiamo che $Nemico(Nono, America)$ quindi $Nemico(x, America) \Rightarrow Ostile(x)$ viene soddisfatta con $\{x/Nono\}$ e viene aggiunto $Ostile(Nono)$
5. per finire: $American(x) \wedge Arma(y) \wedge Vende(x, y, z) \wedge Ostile(z) \Rightarrow Criminale(x)$ è soddisfatta con la sostituzione $\{x/West, y/M_1, z/Nono\}$ e $Criminale(West)$ viene aggiunto.

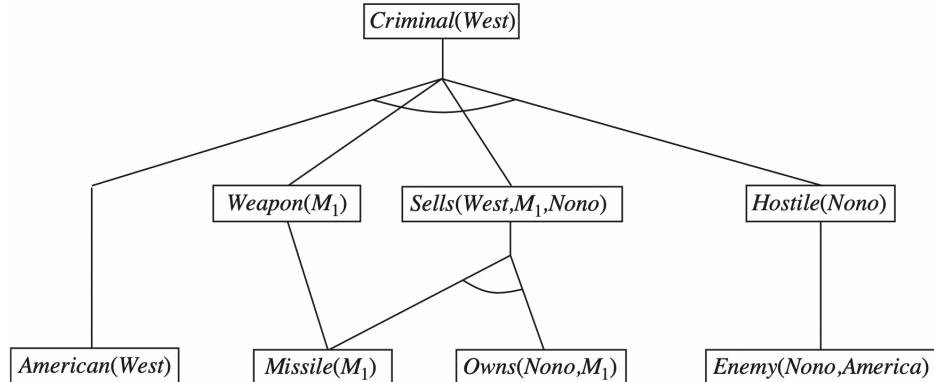


Figure 11: Processo di dimostrazione in avanti

5.17.2 Analisi di FOL_FC_Ask

Il processo è corretto perché il Modus Ponens generalizzato è corretto. È completo per KB di clausole Horn definite. Diventa completo e convergente per calcolo proposizionale e per KB di tipo DATALOG (senza funzioni) perché la chiusura deduttiva è un insieme finito. Completo anche con funzioni, ma il processo potrebbe non terminare (il problema diventa semidecidibile).