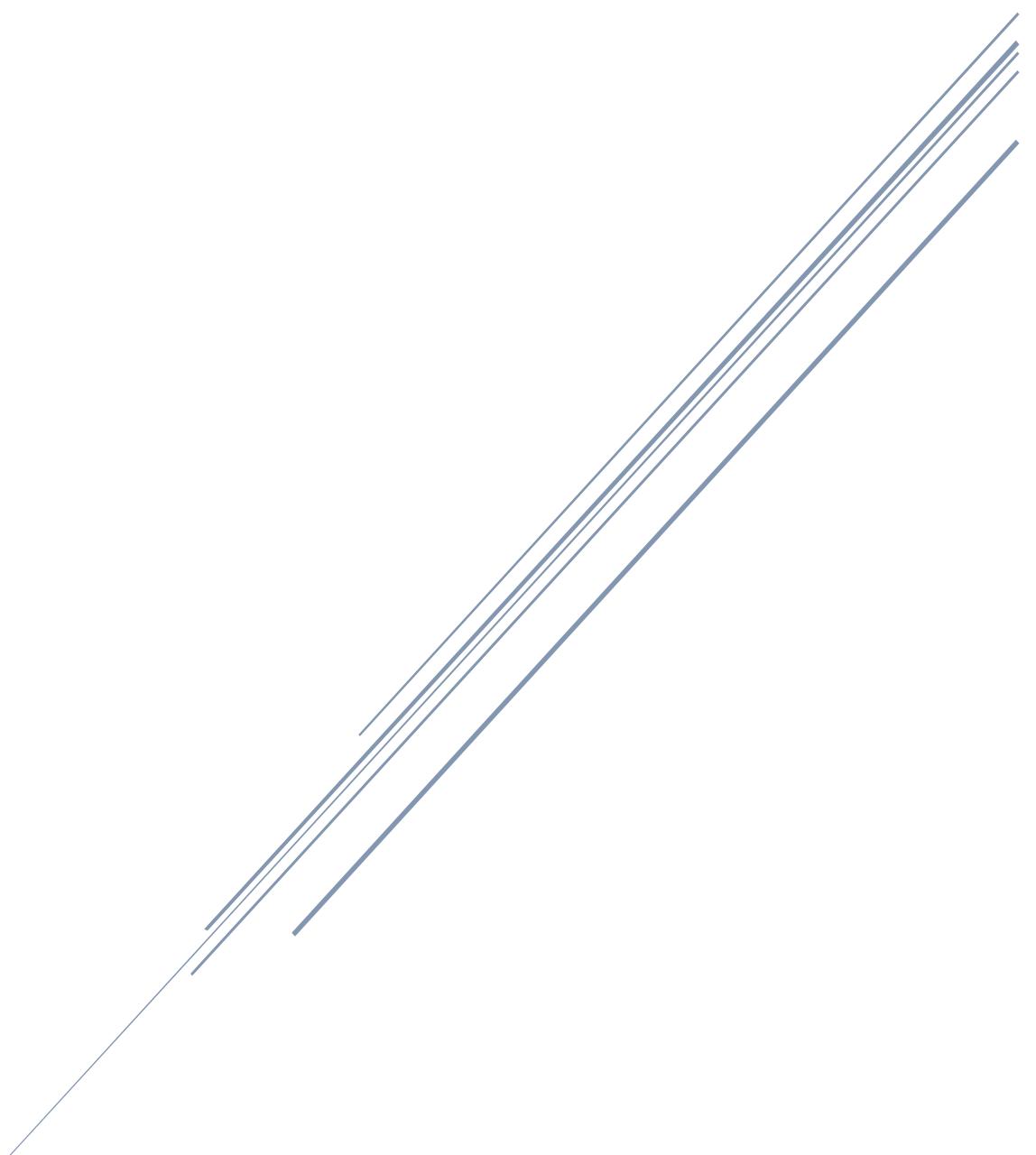


INTELLIGENZA ARTIFICIALE M

Prof. Mello – A.A. 2018/2019



Laura Gruppioni

Sommario

1 – INTRODUZIONE	6
1.1 - DIVERSI TIPI DI INTELLIGENZA.....	6
1.2 - TURING E LE MACCHINE INTELLIGENTI.....	6
1.3 - INTELLIGENZA ARTIFICIALE DEBOLE E FORTE.....	6
1.4 - NASCITA E DEFINIZIONE	7
1.5 - DUE APPROCCI.....	7
1.6 - INTRODUZIONE ALLA LOGICA	7
1.7 - LINGUAGGI DICHIARATIVI E PROLOG	7
1.8 - APPRENDIMENTO	8
1.8.1 - RETI NEURALI.....	8
1.8.2 - DEEP LEARNING	8
1.8.3 - INTELLIGENZA COLLETTIVA	8
1.9 - APPLICAZIONI	9
1.9.1 - SEMANTIC WEB	9
1.9.2 - MACHINE LEARNING	9
1.10 - PROPRIETÀ DELL'AI.....	9
1.10.1 - AGENTI.....	10
1.11 - RIFLESSIONI SOCIALI ED ETICHE	10
2 – SISTEMI BASATI SULLA CONOSCENZA	11
2.1 - PROGRAMMA = CONOSCENZA + CONTROLLO	11
2.1.1 - Principi architetturali	11
2.2 - UNA NUOVA MACCHINA VIRTUALE.....	11
2.3 - SISTEMA DI PRODUZIONI	12
2.4 - ARCHITETTURA GENERALE	13
2.5 - DUE APPROCCI.....	13
3 – SPAZIO DEGLI STATI	14
3.1 - AGENTI RAZIONALI	14
3.2 - RICERCA IN UNO SPAZIO DEGLI STATI	14
3.3 - FORMULAZIONE DEL PROBLEMA.....	15
3.4 - ALTRE CARATTERISTICHE DEL PROBLEMA.....	15
4 – STRATEGIE DI RICERCA.....	16
4.1 – INTRODUZIONE ALLE STRATEGIE	16
4.1.1 - STRUTTURE DATI PER L'ALBERO DI RICERCA.....	16
4.1.2 - STRATEGIE DI RICERCA	17

4.2 - STRATEGIE DI RICERCA NON INFORMATE	17
4.2.1 - BREADTH-FIRST	18
4.2.2 - DEPTH-FIRST	18
4.2.3 - RICERCA A PROFONDITÀ LIMITATA.....	19
4.2.4 - RICERCA AD APPROFONDIMENTO ITERATIVO o IDS.....	19
4.3 - CONFRONTO FRA LE STRATEGIE DI RICERCA NON INFORMATE.....	20
4.4 – STRATEGIE DI RICERCA INFORMATE	20
4.4.1 - RICERCA BEST-FIRST	21
4.4.2 – GREEDY	22
4.4.3 - A*	23
4.4.4 - FUNZIONI EURISTICHE.....	24
4.4.5 - ALGORITMI COSTRUTTIVI E DI RICERCA LOCALE.....	25
4.4.6 - META EURISTICHE	26
4.4.7 - RICERCA IN GRAFI AND/OR.....	27
4.4.8 - ALTRE TECNICHE EURISTICHE.....	27
5 – PROBLEMI DI SODDISFACIMENTO DI VINCOLI.....	29
5.1 - SODDISFACIMENTO DI VINCOLI	29
5.1.1 - TIPI DI VINCOLI	29
5.2 - TIPI DI CSP	29
5.2.1 - ALBERO DECISIONALE	30
5.3 - APPROCCI RISOLUZIONE CSP.....	30
5.4 - ALGORITMI GENERATIVI/SENZA PROPAGAZIONE	30
5.4.1 – GENERATE AND TEST	31
5.4.2 – STANDARD BACKTRACKING	31
5.5 - ALGORITMI DI PROPAGAZIONE	31
5.5.1 – FORWARD CHECKING	32
5.5.2 – LOOKING AHEAD.....	32
5.6 – CLASSIFICAZIONE DELLE EURISTICHE	33
5.7 – TECNICHE DI CONSISTENZA	34
5.7.1 - CONSTRAINT GRAPH	34
5.7.2 – GRADI DI CONSISTENZA	34
5.7.3 - CONSTRAINT SOLVER	37
5.8 - CSP E PROBLEMI DI OTTIMIZZAZIONE	38
5.9 – TECNICHE ALTERNATIVE	38
6 – GIOCHI	39
6.1 – ALGORITMO MIN-MAX.....	39

6.1.1 – PROPRIETÀ DI MIN-MAX	40
6.2 – ALGORITMO MIN-MAX RIVISTO.....	40
6.3 – TAGLI ALFA BETA.....	41
6.4 – ALGORITMO ALFA-BETA	43
6.4.1 - EFFICACIA DEI TAGLI.....	43
7 – INTRODUZIONE ALLA LOGICA.....	45
7.1 - LOGICA PREDICATI PRIMO ORDINE	46
7.1.1 - ALFABETO	46
1.1.2 - SINTASSI (Prolog).....	46
7.2 - SEMANTICA.....	47
7.2.1 - INTERPRETAZIONE.....	48
7.2.2 – VALORE DI VERITÀ.....	48
7.2.3 - MODELLI	48
7.2.4 - INSIEMI DI FORMULE	48
7.2.5 - CONSEGUENZA LOGICA.....	49
7.2.6 – SISTEMI DI REFUTAZIONE.....	49
7.3 – TEORIE DEL PRIMO ORDINE.....	49
7.3.1 – REGOLE DI INFERNZA.....	50
7.4 - DEDICIBILITÀ	50
7.5 – CORRETTEZZA E COMPLETEZZA.....	50
7.6 – MONOTONICITÀ.....	51
7.6.1 - PROPRIETÀ.....	51
8 - INFERNZA E LOGICA PROPOZIZIONALE	52
8.1 - CLAUSOLE.....	52
8.2 - PRINCIPIO DI RISOLUZIONE.....	52
8.2.1 - DIMOSTRAZIONE	52
8.3 - ALGORITMO DI RISOLUZIONE PER LOGICA PROPOZIZIONALE	53
8.4 - FORWARD CHAINING E BACKWARD CHAINING	53
9 - INFERNZA E LOGICA DEI PREDICATI	54
9.1 - TRASFORMAZIONE IN CLAUSOLE.....	54
9.1.1 – TRASFORMAZIONE IN FBF CHIUSA	54
9.1.2 – APPLICAZIONE EQUIVALENZE PER CONNETTIVI LOGICI.....	54
9.1.3 – APPLICAZIONE NEGAZIONE AD ATOMI E NON A FORMULE COMPOSTE	54
9.1.4 – CAMBIAMENTO NOMI VARIABILI	55
9.1.5 – SPOSTAMENTO DEI QUANTIFICATORI.....	55
9.1.6 – FORMA NORMALE CONGIUNTIVA	55

9.1.7 - SKOLEMIZZAZIONE	55
9.1.8 – ELIMINAZIONE DEI QUANTIFICATORI UNIVERSALI	55
9.2 - UNIFICAZIONE.....	56
9.3 – SOSTITUZIONI E RENAMING.....	56
9.4 - SOSTITUZIONE UNIFICATRICE	56
9.5 - ALGORITMO DI UNIFICAZIONE.....	57
9.6 - OCCUR CHECK	57
9.7 - PRINCIPIO DI RISOLUZIONE PER CLAUSOLE GENERALI	58
9.7.1 – MGU PER RICAVARE IL RISOLVENTE	58
9.8 - CORRETTEZZA E COMPLETEZZA	59
9.9 - STRATEGIE.....	59
9.9.1 - STRATEGIA LINEARE	60
9.9.2 - STRATEGIA LINEAR-INPUT.....	60
9.10 - CLAUSOLE DI HORN	61
9.11 – GMP: MODUS PONENS GENERALIZZATO.....	61
10 - PROLOG	62
10.1 - INTRODUZIONE AL PROLOG	62
10.1.1 - INTERPRETAZIONE DICHIARATIVA.....	63
10.1.2 – RISOLUZIONE SLD	64
10.1.3 - UNIFICAZIONE.....	64
10.1.4 - OCCUR CHECK	65
10.1.5 - DERIVAZIONE SLD	65
10.1.6 - NON DETERMINISMO.....	66
10.1.7 - ALBERI SLD	67
10.1.8 – PROLOG	68
10.2 – ARITMETICA E RICORSIONE	70
10.2.1 – IL PREDICATO IS	70
10.2.2 – TERMINI, OPERATORI ED ESPRESSIONI	71
10.2.3 – CALCOLO DI FUNZIONI.....	71
10.2.4 – RICORSIONE E ITERAZIONE	71
10.3 – LISTE	73
10.3.1 – OPERAZIONI SULLE LISTE.....	73
10.4 – CUT	76
10.4.1 – MODELLO RUNTIME DI PROLOG	76
10.4.2 – EFFETTI DEL CUT.....	76
10.4.3 – MUTUA ESCLUSIONE TRA CLAUSOLE	78

10.5 – LA NEGAZIONE	78
10.5.1 – NEGAZIONE PER FALLIMENTO	79
10.5.2 – RISOLUZIONE SLDNF	79
10.5.3 – NEGAZIONE E QUANTIFICATORI	80
10.5.4 – NEGAZIONE IN PROLOG	80
10.6 – META-PREDICATI	80
10.6.1 – PREDICATO CALL.....	81
10.6.2 – PREDICATO FAIL.....	81
10.6.3 – PREDICATI SETOF E BAGOF.....	82
10.6.4 – PREDICATO FINDALL.....	83
10.7 – META-INTERPRETI.....	83
10.7.1 – ACCESSO ALLE CLAUSOLE	83
10.7.2 – MODIFICHE AL DB.....	83
10.7.3 – META-INTERPRETI	84
10.7.5 – META-INTERPRETE FORWARD	84
10.8 – VINCOLI IN PROLOG	85
10.8.1 – CONSTRAINT LOGIC PROGRAMMING (CLP)	86
11 - PLANNING	87
11.1 - RAPPRESENTAZIONI	87
11.1.1 – RAPPRESENTAZIONE DELLO STATO	87
11.1.2 – RAPPRESENTAZIONE DEL GOAL	88
11.1.3 – RAPPRESENTAZIONE DELLE AZIONI	88
11.2 - PIANIFICAZIONE	88
11.2.2 – PIANIFICAZIONE CLASSICA	88
11.2.3 - PLANNING DEDUTTIVO	89
11.2.4 – PLANNING MEDIANTE RICERCA	90
11.2.5 – PIANIFICAZIONE IN PRATICA	93
11.3 - ESECUZIONE	93
11.3.1 – PIANIFICAZIONE REATTIVA	94
11.3.2 - SISTEMI REATTIVI PURI	94
11.3.3 - PIANIFICATORI IBRIDIDI	94

1 – INTRODUZIONE

Difficile definire con precisione cosa sia la disciplina dell'intelligenza artificiale, **non c'è una definizione non ambigua di intelligenza**. Essa **non è misurabile**.

1.1 - DIVERSI TIPI DI INTELLIGENZA

- **Metafora dei sistemi evolutivi:** da specie a specie c'è una evoluzione che permette loro di sopravvivere all'ambiente e al cambiamento dell'ambiente
- **Percezione, Apprendimento, Ragionamento, Astrazione:** ho bisogno di percepire l'ambiente, di apprendere, di ragionare (problem solving) e ho bisogno anche di astrarre per avere applicazioni più generali.
- Esiste un concetto di intelligenza che è quello dell'**intelligenza collettiva**: noi pensiamo ad una singola entità come ad un robot, ad esempio, ma abbiamo anche sistemi che singolarmente hanno potenzialità ridotte, ma insieme sono una potenza. **Un agente intelligente che non sa risolvere solo formalmente sistemi matematici/logici, ma interagisce con l'ambiente.** È diverso da risolvere un problema di logica: se pensiamo alle macchine intelligenti che si guidano da sole, l'ambiente gioca un ruolo fondamentale. La macchina deve essere in grado di compiere delle azioni che devono essere in grado di modificare l'ambiente con cui sto lavorando. È una declinazione molto diversa che richiede diversi approcci (formale e simbolico).
- **Logica: la prima parte dell'intelligenza perché è formalizzabile.**

1.2 - TURING E LE MACCHINE INTELLIGENTI

- **Gioco dell'imitazione:** si parla con una macchina e le si fanno domande per capire se dall'altra parte c'è effettivamente un essere umano. Se non si capisce con certezza, allora si ha a che fare con una macchina intelligente.
- **Test di turing (1950):** è sostanzialmente un foglietto scritto in linguaggio naturale, l'interfaccia è limitata. Oggi quel test dovrebbe avere componenti di un sistema con interfaccia grafica (non con solo terminale) se non addirittura umano. **Si parla di test di turing universale perché deve avere tutte le caratteristiche di un essere umano per poter ingannare l'interrogante.** (es. chatbot)
- Figli di **Eliza**: pattern in cui in base alla domanda, si daranno determinate risposte. L'interazione è tramite questi pattern: dopo un po' con Eliza si capiva che ripeteva le stesse cose e non si ricordava chissà cosa, era molto schematica. **Oggi abbiamo SIRI, Cortana, Alexa.**
- **Schemi di Winograd:** sottolinea quanto sia difficile anche solo interpretare in modo completo una frase o una parola. Ci sono ragionamenti semantici che per un umano sono semplici, ma per una macchina assolutamente no: se Giovanna ringrazia Maria, si suppone che sia lei ad aver ricevuto il regalo. **I sistemi di logica, se noi scriviamo gli assiomi, tirano fuori tutto quello che serve. Il problema è trasferire la conoscenza semantica, il significato.** È importante la rappresentazione della conoscenza ed è diventata fondamentale per fare davvero la differenza.
- **Searle e la stanza cinese (1980):** il test di Turing è stato molto criticato anche perché il sistema capisce davvero quello che sta facendo? Supponiamo di avere nella stanza una persona che sa solo inglese, abbiamo un'interfaccia che è solo in cinese. Immaginiamo che ci sia un pattern che traduce sintatticamente frasi cinesi in inglese. Poi si produce una risposta che con un pattern inverso traduce dall'inglese al cinese la risposta. Il sistema dà un'illusione.

1.3 - INTELLIGENZA ARTIFICIALE DEBOLE E FORTE

- **AI DEBOLE:** è possibile costruire macchine in modo che agiscano COME SE fossero intelligenti?
- **AI FORTE:** è possibile costruire macchine che PENSINO intelligentemente? Queste macchine devono rendersi conto di quello che stanno facendo

L'intelligenza artificiale è altamente interdisciplinare: filosofia, logica, matematica, economia, psicologia, informatica, ingegneria, linguistica...

1.4 - NASCITA E DEFINIZIONE

Nata nel 1956 da Minsky, McCarthy, Shannon, Newell, Simon. **Alcune definizioni:**

- È lo studio di come far fare ai calcolatori cose che, ora come ora, gli esseri umani fanno meglio
- È la costruzione di un computer che è in grado di soddisfare il test di Turing (ragionamento, linguaggio naturale, apprendimento). Se Totale (situato in un ambiente) anche percezione, visione, movimento, robotica.
- Altre definizioni di IA tendono a non legare necessariamente l'intelligenza (artificiale) agli umani e sottolineano l'interazione con il mondo esterno e le capacità di adattarsi ad esso. Anche gli animali, i vegetali, e le macchine possono essere intelligenti se riescono ad interagire in modo utile con l'ambiente che li circonda.

1969–1979: sistemi basati sulla conoscenza. Costruzione di sistemi a regole che avevano comportamento simile ad un esperto in quel settore. **Allora si chiamavano Sistemi Esperti** e si pensava che questo tipo di sistemi potessero sfociare in sistemi automatici che potessero sostituire umani

1988-2000: Web e l'era di internet. Ossigeno all'IA perché c'era la possibilità di accedere a moltissime informazioni, che sono alla base della conoscenza.

1.5 - DUE APPROCCI

- **TOP DOWN o SIMBOLICO**
 - Stati mentali identificati con rappresentazioni di tipo simbolico
 - Sistema simbolico fisso con simboli combinati in strutture
 - Concetti descritti tramite le loro proprietà o il loro modo di essere rappresentati
- **BOTTOM UP o CONNESSIONISTA**
 - Reti di neuroni artificiali
 - Concetti appresi in modo implicito a partire da esempi

1.6 - INTRODUZIONE ALLA LOGICA

- **Ragionamento deduttivo:** implicazione logica, non si impara nuova conoscenza, corretta
- **Ragionamento induttivo:** implicazione, si impara ma si può non avere correttezza
- **Ragionamento ipotetico o abduttivo:** risale alle cause mediante l'osservazione
- **Ragionamento per analogia:** usa il principio di somiglianza, va mediato

1.7 - LINGUAGGI DICHIARATIVI E PROLOG

Sappiamo che i linguaggi possono essere:

- Imperativi (C, Python...)
- Ad oggetti (Java, C++)
- Dichiаративи (Prolog)
- Funzionali (Lisp)

Algoritmo = Logica + Controllo

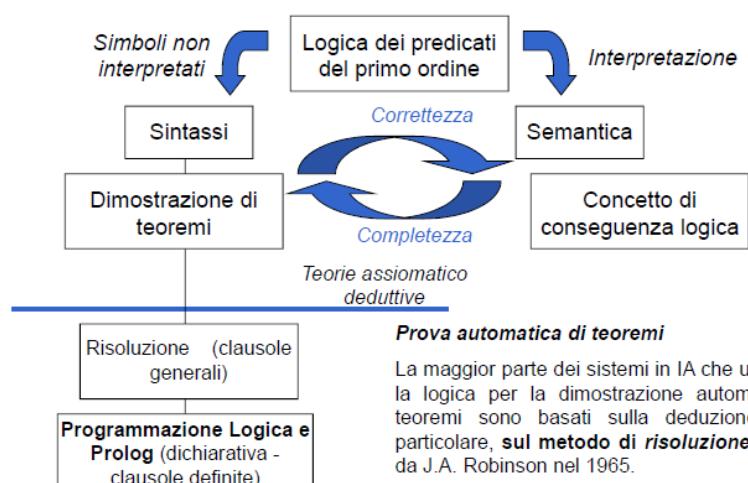
Logica = conoscenza sul problema determina la correttezza

Controllo = strategia risolutiva che ne determina l'efficienza

PROgramming in LOGic = Prolog.

È il più noto linguaggio di programmazione logica e si fonda sulle idee di Kowalski. Prima realizzazione nel 1973 da parte di Colmarchur.

Un programma Prolog è un insieme di clausole di Horn che rappresentano fatti, regole e goal.



Prova automatica di teoremi

La maggior parte dei sistemi in IA che utilizzano la logica per la dimostrazione automatica di teoremi sono basati sulla deduzione e, in particolare, sul metodo di **risoluzione** definito da J.A. Robinson nel 1965.

1.8 - APPRENDIMENTO

Forme di apprendimento più usate:

- **Apprendimento supervisionato**
 - Si parte da un training set fornito da un insegnante
 - Si usa per problemi di classificazione
- **Apprendimento non supervisionato**
 - Mediante osservazione e scoperta
 - Clustering
- **Apprendimento mediante rinforzo**
 - Si apprende in base ad esperienze passate
 - Si osserva in modo critico un risultato buono o cattivo e si modifica il comportamento
 - Molto usato in robotica

Il metodo induttivo si applica anche a sistemi simbolici, quindi quando si parla di apprendimento può essere fatto con reti neurali o generando sistemi come alberi decisionali o sistemi con regole.

Ci sono casi in cui è meglio usare un approccio sub-simbolico o altri in cui è meglio usare deep e machine learning per capire che modello è stato costruito.

1.8.1 - RETI NEURALI

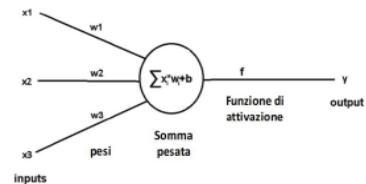
Primo modello matematico di neurone artificiale ispirato ai neuroni biologici fu proposto nel 1943 da McCulloch e Pitts.

Un neurone riceve un insieme di ingressi, ne fa una somma pesata ed applica poi una funzione di attivazione per calcolare l'uscita. L'uscita è controllata da una funzione di attivazione: ogni neurone si attiva soltanto nei casi in cui il proprio ingresso supera una certa soglia.

Il potere di reti neurali è il fatto di connettere tanti neuroni: la potenza di queste reti nasce dalla possibilità di inserire delle strutture e degli stati nascosti (neuroni non connessi direttamente a input/output) che lavorano in modo molto efficiente.

Ad esempio sono molto utili per il riconoscimento di testi scritti a mano in cui un singolo simbolo può essere scritto in modo diverso in base alla grafia di ogni persona (**MNIST dataset**).

Problema: il percepitrone può rappresentare solo funzioni lineari. Se si considerano reti multi-strato aumenta il potere espressivo. Ma come si configura una rete di neuroni in modo efficiente? Esistono diverse architetture.



1.8.2 - DEEP LEARNING

Quando si parla di deep learning, si parla sempre di reti neurali ma si pone l'accento sulla grande mole di dati che si hanno a disposizione (**big data**) e sulla grande potenza di calcolo dei computer moderni.

DNN: Deep Neural Networks hanno un numero di livelli compreso tra 7 e 50.

Qui si può rappresentare il mondo con una gerarchia di concetti, ad esempio un corpo ha un viso che a sua volta ha occhi, naso e sopracciglia.

1.8.3 - INTELLIGENZA COLLETTIVA

La natura ha sviluppato tecniche intelligenti per:

- La difesa dell'organismo, la selezione della specie (algoritmi genetici)
- La coordinazione tra animali sociali (termiti, formiche... **Swarm Intelligence**)

Es. Le formiche, se noi mettiamo del cibo, prima hanno un comportamento più o meno casuale, ma dopo un po' si avvicinano sempre di più con strade diverse. Dopo un altro po' quasi tutte seguono un percorso, una fila, che sembra quello preferito. È dimostrato che quel percorso è il percorso migliore. Si è scoperto che le formiche comunicano lasciando sul percorso un odore. Questo ormone con il tempo tende a diminuire fino a scomparire, quindi perché scelgono la strada a migliore costo? È quella in cui evapora meno, quindi ha una vita più lunga.

1.9 - APPLICAZIONI

Le più note applicazioni dell'AI sono:

- Sistemi esperti/di supporto alle decisioni (medicina, domotica, sostenibilità)
- Sistemi formali e giochi (scacchi/go)
- Linguaggio naturale (giochi a quiz, chatbot)
- Visione (riconoscimento di immagini)
- Robotica e sistemi autonomi (robot, auto con pilota automatico)

MYCIN: creato a metà degli anni 70 è molto conosciuto perché è stato uno dei primi. Doveva riconoscere se c'era infusione nel sangue e dare opportuna terapia antibiotica. È simbolico, basato su regole. C'è un fattore di incertezza.

I sistemi esperti e i sistemi knowledge-based hanno dei problemi: bisogna trovare persone esperte che hanno voglia di condividere il loro sapere. Spesso invece un esperto ha poco tempo e magari ha un atteggiamento ostile nel condividere il suo sapere per cose come queste. Formalizzare quello che ci viene detto inoltre non è così semplice. Più una persona è esperta di un settore, più la presa delle decisioni è quasi intuitiva: a volte è preferibile parlare con persone che hanno esperienze più scolastiche e non intuitive.

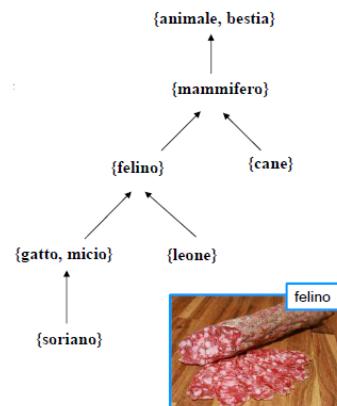
Ecco perché la conoscenza di un essere umano non va sostituita, ma coadiuvata. Servono tecniche di apprendimento e data-mining.

1997 Deep Blue, computer dell'IBM, sconfigge Kasparov a scacchi.

2011 Watson, computer dell'IBM, batte i campioni in carica al gioco a quiz televisivo Jeopardy.

2016 AlphaGo batte il campione in carica nel gioco del Go.

ImageNet Challenge: margine di errore in sistemi di AI nel riconoscimento e classificazione di immagini passa dal 26% al 3%.



1.9.1 - SEMANTIC WEB

Si ragiona su tutti i dati presenti sul Web.

Se cerco semplicemente qualcosa che mi parli di un "felino", si fa una semplice ricerca per chiave e si ottengono tutti i felini. Servono dei link in cui è stabilita una gerarchia, serve un'organizzazione, così ho sistemi più intelligenti. Ma felino cos'è? Felino può essere un animale o un salame. Consente di fare anche **disambiguazione semantica di parole nel linguaggio naturale (BabelNet)**.

1.9.2 - MACHINE LEARNING

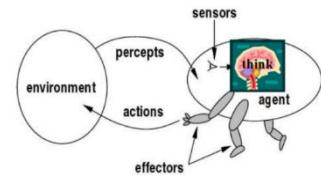
Può essere la tecnica vincente per **estrarre informazioni utili da molti dati**, ad esempio in ambito medico per previsioni o ipotesi.

1.10 - PROPRIETÀ DELL'AI

1. **FAIRNESS**: decisioni non devono essere discriminatorie.
2. **TRANSPARENCY**: sistema usa un modello comprensibile all'uomo in modo che ne possa capire e previsto il comportamento in ogni situazione.
3. **VERIFYABILITY**: si può dimostrare formalmente che il sistema è corretto rispetto ad alcune proprietà.
4. **EXPLAINABILITY**: si devono fornire motivazioni di una scelta o di una conclusione.
5. **ACCOUNTABILITY**: responsabilità per le decisioni prese.
6. **ACCURACY, PRIVACY...**

Attenzione perché in base ai dati, le macchine possono imparare a fare scelte discriminatorie a priori. Spesso i dati non sono espliciti, sono nascosti. Es. macchina assegnava lavoro solo a uomini e non donne. Non c'era scritto "uomo sì e donna no", ma magari guardava le assenze nei lavori precedenti e magari le donne venivano discriminate perché erano state assenti per maternità.

Per cercare di rispettare queste proprietà, l'obiettivo è riconciliare le due anime dell'IA, cioè il deep learning, che eccelle nel livello percettivo e nell'apprendimento, con sistemi simbolici che sono trasparenti e che eccellono in ragionamento ed astrazione.



1.10.1 - AGENTI

Quando mettiamo un agente, che è un **corpo con un cervello e dei sensori in un ambiente**, questo interagisce con l'ambiente. Gli **agenti razionali** sono quelli che osservano il ciclo **observe-think-act (Kowalski)**. Esistono vari esempi di robot: ASIMO, Atlas, Robocup, NAO... Esempi di AI anche nell'arte: scritto un film, composta una canzone, usato su Van Gogh.

1.11 - RIFLESSIONI SOCIALI ED ETICHE

Non ci si può fidare sempre delle DNN perché spesso imparano ma non capiscono, trovano semplicemente delle soluzioni specifiche per i problemi che hanno incontrato senza essere capaci di comprendere i motivi delle loro scelte. **Non distinguono tra causalità e correlazione** (es. carro armato di notte e di giorno).

Il concetto di etica è molto importante perché si ha a che fare con **dati sensibili**, quindi tutto il discorso della **privacy**. È importante essere aware su quello che si sta muovendo sull'approccio e sul tentativo di capire se possono esistere dei regolamenti per governare l'utilizzo dell'IA.

- **Continua sostituzione da parte delle macchine di attività una volta svolte dall'uomo.** Non è una novità (rivoluzione industriale), ma questa volta anche in attività professionalizzanti ritenute appannaggio dell'uomo.
- **Perdita di molti posti di lavoro per la classe classi medio/basse**, mentre **aumento di richiesta per posti ad alta professionalità nel settore ICT**.
- **Rischio di aumento della disoccupazione, povertà e della disuguaglianza.** Ma è colpa dell'IA o dei modelli economici/politici adottati?

Cosa può fare l'IA al riguardo? Non subiamola, ma invece utilizziamola per ottenere applicazioni con impatti positivi e profondi sulla società e l'economia. Droni armati, auto con guida autonoma, sofisticati sistemi di AI che guadagnano in Borsa evolvendosi sulle conoscenze acquisite ecc. Chi ha la **responsabilità delle violazioni che possono essere compiute** attraverso i loro utilizzo? Sono necessarie delle **regole**:

- **Regolamento sull'utilizzo delle armi autonome.**
- **Obbligo per le applicazioni di intelligenza artificiale di spiegare il motivo di una decisione** (Regolamento generale sulla protezione dei dati, 2018, Comunità europea).
- **Cross Industry collaboration** (Google, Apple, Amazon, IBM, Microsoft).

Roboetica: il comportamento etico dei robot dipende strettamente da quanto richiesto e realizzato dal progettista. Man mano che cresce l'autonomia dei robot nel prendere decisioni a fronte di eventi inaspettati, senza alcun intervento umano si parla di etica dei robot intesi come entità decisionali autonome.

Le tre leggi della robotica (Asimov):

1. Un robot non può recare danno a un essere umano, né può permettere che, a causa del suo mancato intervento, un essere umano riceva danno.
2. Un robot deve obbedire agli ordini impartiti dagli esseri umani, purché tali ordini non contravvengano alla prima Legge.
3. Un robot deve proteggere la propria esistenza, purché questa autodifesa non contrasti con la prima e seconda Legge.

E una quarta, superiore, “**legge zero**”: Un robot non può recar danno all'umanità e non può permettere che, a causa di un suo mancato intervento, l'umanità riceva danno.

2 – SISTEMI BASATI SULLA CONOSCENZA

2.1 - PROGRAMMA = CONOSCENZA + CONTROLLO

Il programma è un ambiente in cui si rappresenta, si utilizza e si modifica una base di conoscenza.

Potrei usare lo stesso controller, lo stesso formalismo di rappresentazione della conoscenza, per fare diverse cose. Posso mantenere la stessa KB e creare controller diversi, anche per lo stesso tipo di sistema.

2.1.1 - Principi architetturali

Ogni sistema basato sulla conoscenza deve riuscire ad esprimere due tipi di conoscenza:

- Conoscenza sul dominio dell'applicazione (cosa)
- Conoscenza sull'utilizzo della conoscenza per risolvere problemi (come, controllo)

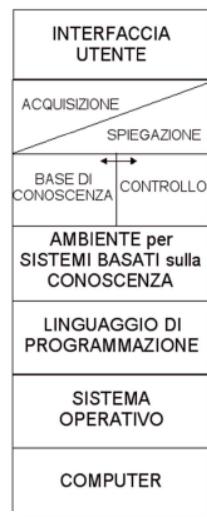
Se si tengono separati questi tipi di conoscenza, si ha alta modularità e una programmazione dichiarativa. Inoltre in questo modo la conoscenza sul problema è espressa indipendentemente dal suo utilizzo: l'ambiente, ad esempio, non è un problema perché l'algoritmo viene eseguito sia in un ambiente che in un altro.

Ovviamente ci sono delle problematiche di:

- Rappresentazione della conoscenza
- Organizzazione del problema
- Decisione delle strategie di controllo

Ci sono però dei vantaggi nell'uso dell'approccio dichiarativo:

- Si imposta il problema ad alto livello
- Si può usare la stessa conoscenza in diversi modi, ottenendo generalità
- Si può modificare facilmente poiché si ha flessibilità



2.2 - UNA NUOVA MACCHINA VIRTUALE

Esempio su un problema di diagnostica: bisogna prescrivere una medicina in base ai risultati di un esame di laboratorio.

FATTI:

```
gram(neg).      -> esami del sangue negativi  
not(allergic(antb)). -> non allergico ad antibiotici
```

REGOLE:

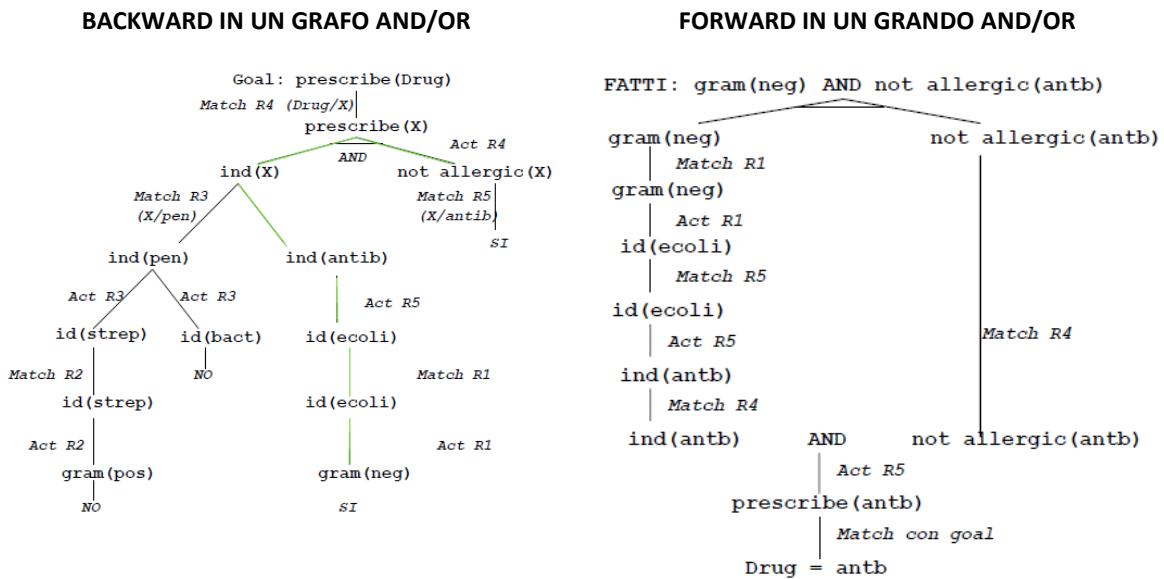
- R1: `gram (neg) → id (ecoli)`.
Se il risultato dell'esame è *gram-negativo* allora l'identità è *enterium-colis*
- R2: `gram (pos) → id (strep)`.
Se il risultato dell'esame è *gram-positivo* allora l'identità è *streptococco*
- R3: `id(strep) OR id(bact) → ind(pen)`.
Se l'identità è *streptococco* o *bacterio* allora è bene indicare penicillina
- R4: `ind(X) AND not (allergic(X)) → prescribe(X)`.
Se è bene indicare una certa medicina e il paziente non è allergico a tale medicina, allora si può prescrivere tale medicina al paziente
- R5: `id(ecoli) → ind(antb)`.
Se l'identità è *enterium-colis* allora è bene indicare antibiotici

Qui ho due approcci: o parto dai fatti e arrivo all'obiettivo, oppure parto dall'obiettivo e guardo le regole all'indietro.

Es. Parto dal goal e prendo le regole che hanno come conseguente qualcosa che unifichi/faccia match con quello e si creano dei sotto-goal che sono gli antecedenti. Quella regola deriverebbe il conseguente se fossero veri gli antecedenti. Questo si fa finché non raggiungo un goal che è anche un fatto. A quel punto, quel sotto-goal è dimostrato e ho la mia catena all'indietro.

Differenza tra controllo backward o forward: nel caso reale, ad esempio, se usassimo il forward per andare dal medico, succederebbe che "guardi dottore non sto tanto bene blabla" e il medico ci dice "bene, questi sono gli nmila esami che devi fare, buon divertimento". Risultato: -.-"

In alcuni casi si lavora prima in forward poi in backward: ad es. il medico fa per ipotesi e poi va in backward a verificare che le ipotesi fossero corrette.



Nel primo caso (backward) ho un fatto: avrò un gram negativo quindi entro nella regola 1. Dato che noi siamo andati nella r1, r3 si scarta perché non è nel nostro ramo, quindi andremo sulla ind(antib). Derivando quello, otteniamo che il paziente non è allergico all'antibiotico (qui abbiamo un fatto che dice che non lo è), allora facendo AND, ottengo che verrà descritto l'antibiotico.

2.3 - SISTEMA DI PRODUZIONI

Il modo di concepire il problema cambia in base a quello che dobbiamo fare. Noi costruiamo questi sistemi impartendo loro un'opportuna conoscenza sul problema e progettando la parte di inferenza, di controllo. Allora i nostri sistemi diciamo che sono composti da una base di conoscenza e da un motore di inferenza o controllo.

L'architettura che ci è stata presentata va spesso sotto il nome di sistemi di produzioni perché **ci sono degli operatori che lavorano all'interno di uno stato e lo modificano** e quindi si hanno **sistemi rule based**.

Nella memoria di lavoro andiamo ad inserire i nostri stati intermedi. Gli operatori sono le regole che sono nella base di conoscenza. Le regole applicabili sono quelle per cui le precondizioni sono in qualche modo verificate all'interno della memoria di lavoro. Quindi **all'inizio in memoria guardo i fatti e vedo se le regole sono applicabili, queste regole tipicamente saranno istanze di regole con degli elementi specifici e non saranno regole generali. Da una regola, posso avere tante istanze di regole.**

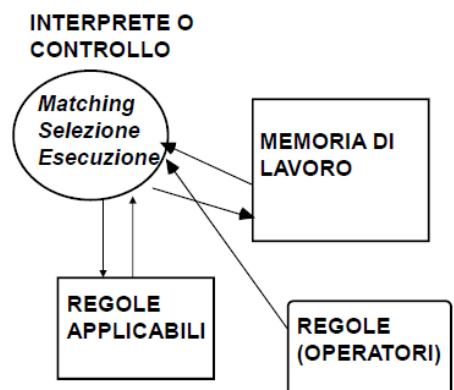
Es. R1: se $a(x)$ allora $b(x)$

Nella mia memoria di lavoro ho a(1), a(2) e a(3): la memoria di lavoro in questo momento allora userà delle istanze della regola r1. Il motore di inferenza deve trovare gli operatori/le istanze di r1 in cui x è legato a 1, a 2 o a 3 (tre istanze). Il sistema costruisce quella che si chiama un'agenda: sono le regole applicabili. Una volta che il sistema le applica, le seleziona e guarda cosa succede. Quindi gli operatori aggiungeranno b(1), b(2) e b(3) fino a raggiungere l'obiettivo.

Nell'agenda si applicano delle politiche di priorità: quali regole vanno applicate.

Il principio è molto diverso rispetto a quello classico perché il sistema attiva le regole in base ai dati.

Definizione: i sistemi di produzione di regole sono programmi che realizzano metodi di ricerca per problemi rappresentati come spazio degli stati.



Sono composti di:

- **Un insieme di regole**
- **Una working memory** che contiene gli stati correnti raggiunti
- **Una strategia di controllo per selezionare le regole** da applicare agli stati della working memory (verifica di precondizioni, matching, test sullo stato goal se raggiunto)

2.4 - ARCHITETTURA GENERALE

Il controllo ha più fasi:

- **Matching:** trova regole applicabili, cioè quelle che vanno messe in agenda
- **Selezione:** qui c'è politica, strategia. Quale regola scelgo? Quale regola applico? Questa parte ha un impatto sul sistema, es. Sistemi forward sono usati in monitoraggio perché regole e operatori non sono tutti uguali. Se faccio monitoraggio probabilmente una situazione di allarme va messa in modo prioritario rispetto ad alcuni controlli normali giornalieri.
- **Esecuzione**

Le regole applicabili sono istanze delle regole generali.

2.5 - DUE APPROCCI

La prima scelta architettonica da fare quando creo un sistema è scegliere la modalità di ragionamento:

- **Forward o data-driven:** nello stato iniziale ci metto i fatti e poi applico le regole di produzione, gli operatori, il cui antecedente fa match con lo stato della memoria di lavoro.

Una volta applicate, le inserisco e termino con successo quando ho raggiunto il goal

- **La memoria di lavoro all'inizio contiene solo i fatti noti**
- **Le regole di produzione applicabili sono quelle il cui antecedente fa match con la memoria di lavoro**
- **Ogni volta che seleziono una regola e la eseguo, nuovi fatti dimostrati sono inseriti nella memoria di lavoro**
- **Si termina quando nella memoria c'è il goal da dimostrare**

- **Backward o goal-driven:** la struttura non cambia dal punto di vista concettuale perché qui applico le regole all'indietro. Faccio matching con il successivo e faccio dei nuovi sotto-goal.

Il tutto termina quando nella memoria di lavoro ci sono solamente dei fatti e in quel caso abbiamo raggiunto il nostro obiettivo.

- **La memoria di lavoro all'inizio contiene il goal**
- **Le regole di produzione applicabili sono quelle il cui conseguente fa match con la memoria di lavoro**
- **Ogni volta che si seleziona ed esegue una regola, nuovi subgoals da dimostrare sono inseriti in memoria**
- **Si termina quando rimangono solo fatti noti**

Esistono anche approcci misti: si combinano i due approcci visti sopra, la memoria è suddivisa in due parti (una contiene i fatti, l'altra i goal e subgoals) e si applicano simultaneamente le regole dei due approcci fino alla terminazione.

3 – SPAZIO DEGLI STATI

3.1 - AGENTI RAZIONALI

In questa prima parte del corpo si usa scrivere degli pseudo-algoritmi che risolvono dei problemi.

Alla fine questo "think" che pensa, lo si può inserire in un agente che potrebbe essere un **agente virtuale o fisico**.

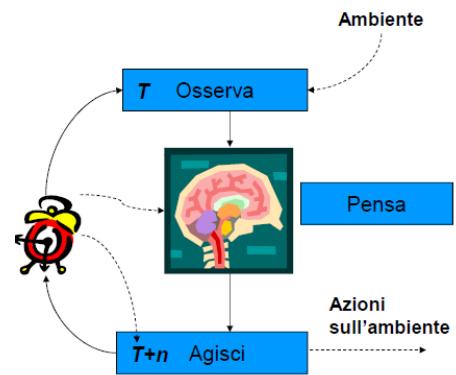
Questo è **pseudocodice interpretato in base al linguaggio**. Noi di questo simple problem solving agent faremo solo il cuore, ma poi la parte percettiva e la parte attiva vanno inserite.

L'agente ha un'interfaccia in cui l'input è la percezione dell'ambiente e l'uscita è un'azione nell'ambiente.

L'agente è un'entità con **stato interno** che è **astratto**, può essere aggiornato dall'attività percettiva.

Es. Spesso gli agenti nello stato hanno le "credenze" (beliefs): il suo stato non è la descrizione del mondo, è come lui percepisce secondo le sue credenze com'è il mondo. Non è una cosa da poco.

Es. Vedo un'immagine, sento un odore, eccetera. Poi c'è un mondo che io mi creo dentro.



```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
    static: seq, an action sequence, initially empty
            state, some description of the current world state
            goal, a goal, initially null
            problem, a problem formulation

    state  $\leftarrow$  UPDATE-STATE(state, percept)
    if seq is empty then do
        goal  $\leftarrow$  FORMULATE-GOAL(state)
        problem  $\leftarrow$  FORMULATE-PROBLEM(state, goal)
        seq  $\leftarrow$  SEARCH(problem)
        action  $\leftarrow$  FIRST(seq)
        seq  $\leftarrow$  REST(seq)
    return action
```

Bisogna formulare il goal e costruirselo. Noi non l'affronteremo specificatamente. Se l'agente è in grado di prefingersi un obiettivo è un agente deliberativo. Generalmente gli obiettivi invece li diamo noi all'agente, quindi non è deliberativo. Poi si formula il problema: formularlo e search sono proprio quello di cui trattiamo noi.

Formulare il problema significa comunicarlo in modo comprensibile alla macchina, formale, per fare in modo che la macchina possa risolverlo.

Per risolverlo, si passa il problema ad un meccanismo di search: i nostri problemi sono fondamentalmente tutti basati sulla ricerca di una soluzione in uno spazio degli stati mio interno, virtuale, che mi sono costruito io. Alla fine questa search mi darà una soluzione che viene formalizzata come una sequenza (in generale) di azioni che voglio svolgere a partire da quello stato.

Ora se ho una sequenza di azioni che risolvono il problema, seleziono la prima e la do in uscita, poi mi tengo in pancia anche le altre perché al giro dopo potrei dover fare le altre azioni.

L'effetto finale di questo schema è che **dato un goal** (noi lo daremo esplicitamente mentre qua è deliberativo) **si costruisce la formulazione del problema e si dà in uscita la prima azione da svolgere.**

3.2 - RICERCA IN UNO SPAZIO DEGLI STATI

Spazio degli stati: è lo spazio composto da tutti gli stati che si possono raggiungere a partire dallo stato iniziale nel momento in cui vogliamo applicare una sequenza di operatori. Sono gli stati raggiungibili. Può essere uno spazio accettabile o di dimensioni molto critiche.

Caratteristiche dello spazio degli stati:

- **Stato iniziale:** significa che l'agente è in quello stato, ma non necessariamente fisico (lo stato iniziale nel caso di backward è lo stato finale)
- **Insieme di azioni** possibili che agiscono sullo stato. Possiamo vederle formalmente come una funzione successore $S(X)$ che genera gli stati che si possono generare. Es. sto camminando, ho 4 possibili stati da generare, la $S(x)$ me li genera tutti e 4: avanti, indietro, destra, sinistra.
- **Un cammino:** sequenza di azioni che conduce da uno stato all'altro

Mi fermo quando raggiungo l'obiettivo. Il test più semplice che possiamo pensare è quello del robot: se trovo posizione che è quella del bar davanti al bancone, allora ho raggiunto il mio obiettivo applicando determinati operatori. (ci sono varie strategie da applicare ovviamente)

Spesso lo stato è descritto in modo un po' più astratto, può essere abbastanza complicato. Ad esempio, supponiamo che lo stato sia determinato in prima battuta dal vedere una certa scena, in questo stato potrebbero esserci informazioni diverse. **Molto spesso lo stato è descritto con delle proprietà** e non in modo maniacale.

Es. in molti casi non mi basta dire solo se posso o non posso raggiungere l'obiettivo, ma voglio sapere anche come posso raggiungerlo. Altre volte vorrei tra le tante soluzioni, anche qual è quella ottima.

Es. Romania con la sequenza di città

3.3 - FORMULAZIONE DEL PROBLEMA

Il problema è definito da quattro punti:

- **Lo stato iniziale**
- **Le azioni o funzioni successore (insieme di coppie azione-stato)**
- **Il goal test** che può essere
 - Esplicito
 - Implicito
- **Il costo della strada** (la somma della distanza, il numero di azioni...)

Una soluzione è una sequenza di azioni che portano dallo stato iniziale al goal.

Es. il puzzle delle 8 tessere, assemblaggio robot, missionari e cannibali, cripto-aritmetica, problema delle N regine, sudoku, torre di Hanoi, commesso viaggiatore, scimmia e banana, capre e cavoli

3.4 - ALTRE CARATTERISTICHE DEL PROBLEMA

Lo spazio di ricerca può essere molto ampio come negli scacchi o nelle parole crociate.

Alcuni sistemi possono essere decomponibili, ma non tutti perché possono esserci dei sotto-problemi interagenti. Es. Equazioni, blocchi cubici.

Per costruire questi sistemi bisogna farsi tante domande: ad esempio dobbiamo chiederci sempre se un'azione che facciamo può invalidare qualcosa che prima era vero. Es. A è sopra b.

Se prendo b, nello stato successivo, questa affermazione non è più vera. Alcuni fatti sono veri in alcuni stati e non in altri: si dice che è una **produzione monotona**. **Se una cosa è dimostrata vera, non può improvvisamente diventare falsa: il sistema può solo crescere, non calare.** Se una cosa è vera, è vera. Non può essere vera in uno stato e falsa nell'altro. Nella modellazione dei blocchi un operatore che prende b crea un nuovo stato in cui b non è più sopra ad a. Abbiamo operatori che invalidano le cose.

I sistemi di logica pura sono monotoni e possono applicare la scomposizione del problema in sotto-problemi.

Es. dimostratore di teoremi

Problema a stati singoli: lo stato è sempre accessibile e l'agente conosce esattamente che cosa produce ciascuna delle sue azioni. Può calcolare esattamente in che stato sarà dopo qualunque sequenza di azioni. (noi tratteremo solo questi)

Problema a stati multipli: lo stato non è completamente accessibile e l'agente deve ragionare su possibili stati che potrebbe raggiungere. L'effetto delle azioni può essere sconosciuto o imprevisto e serve una capacità di rilevamento durante la fase di esecuzione.

4 – STRATEGIE DI RICERCA

4.1 – INTRODUZIONE ALLE STRATEGIE

Il motore di inferenza costruisce lo spazio di ricerca per trovare la soluzione. Supposto di aver formalizzato il problema, ora dobbiamo cercare la soluzione.

Prendiamo la solita metafora del robot che si deve muovere per arrivare dallo stato iniziale ad un goal: abbiamo la funzione successore che dallo stato attuale genera lo stato successivo simulando il "futuro". Da uno stato non tutte le azioni possono essere fatte, quindi avrò degli operatori applicabili e altri no.

Un algoritmo di ricerca prende come input un problema e restituisce una soluzione nella forma di una sequenza di azioni. Una volta che viene trovata la soluzione, le azioni suggerite possono essere realizzate: questa fase è chiamata ESECUZIONE.

Per generare sequenze di azioni è necessario fare più passi:

- **Espansione:** si parte da uno stato e si generano nuovi stati tramite gli operatori. È un'operazione molto costosa quindi va ben calibrata.
- **Strategia di ricerca:** ad ogni passo bisogna scegliere quale stato espandere.
- **Albero di ricerca:** rappresenta l'espansione degli stati a partire dallo stato iniziale (la radice)

```
function TREE-SEARCH(problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
```

4.1.1 - STRUTTURE DATI PER L'ALBERO DI RICERCA

Nell'albero di ricerca si hanno i nodi. L'operatore viene applicato per ottenere un nodo, che può essere il nodo genitore o il nodo figlio e che ha una sua profondità. Il costo del cammino si calcola dallo stato iniziale al nodo scelto.

Nel primo algoritmo, quando ho estratto il nodo, tiro fuori lo stato del nodo e lo metto nel metodo goaltest, cioè controllo se è già il goal. Se ha risultato vero, allora ritorno la soluzione (anche solution è un metodo ma dipende dal problema: se è un planner mi deve tirare fuori la sequenza di azioni, altrimenti basta che mi dica sì oppure no). Se il goaltest fallisce, allora vado nella parte seguente e faccio l'espansione che va inserita nella nuova frontiera.

```
function TREE-SEARCH(problem, fringe) returns a solution, or failure
    fringe  $\leftarrow$  INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node  $\leftarrow$  REMOVE-FRONT(fringe)
        if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
        fringe  $\leftarrow$  INSERT ALL(EXPAND(node, problem), fringe)

function EXPAND(node, problem) returns a set of nodes
    successors  $\leftarrow$  the empty set
    for each action, result in SUCCESSOR-FN[problem](STATE[node]) do
        s  $\leftarrow$  a new NODE
        PARENT-NODE[s]  $\leftarrow$  node; ACTION[s]  $\leftarrow$  action; STATE[s]  $\leftarrow$  result
        PATH-COST[s]  $\leftarrow$  PATH-COST[node] + STEP-COST(node, action, s)
        DEPTH[s]  $\leftarrow$  DEPTH[node] + 1
        add s to successors
    return successors
```

La funzione expand, dato un nodo e la descrizione del problema, deve generare un insieme di nodi che nascono dall'espansione di quel nodo. Dato uno stato, mi dà i successori del mio nodo.

Per ognuno di questi successori devo costruire la struttura nodo perché i successori sono stati. Per ogni successore allora costruisco il suo nodo corrispondente: creo dinamicamente una struttura dati con new.

Il genitore del nodo è il padre di cui mi devo ricordare le azioni (perché se alla fine vuole una sequenza di azioni, devo portarmi dietro la sequenza), poi il valore dello stato del nuovo nodo è quello che mi ha trovato la funzione successore. Nel costo del cammino ci metto il costo del padre più il costo dell'azione che ho applicato (step cost) e come profondità ci metto il padre+1. **Costo tot. ricerca = costo cammino + costo ricerca.**

4.1.2 - STRATEGIE DI RICERCA

I problemi che i sistemi basati sulla conoscenza devono risolvere sono non-deterministici cioè in un certo istante più azioni possono essere svolte (azioni: applicazioni di operatori).

STRATEGIA: informazione sulla conoscenza che sarà applicata potendone invocare molteplici.

Due possibilità:

- **STRATEGIE NON INFORMATE:** non si usa alcuna conoscenza sul dominio, si applicano regole in modo arbitrario (**Ricerca Esaustiva**). Impraticabile per problemi di una certa complessità.
- **STRATEGIE INFORMATE:** si utilizza una Conoscenza Euristica sul problema per la selezione degli operatori applicabili.

La prima non si usa molto ma è basilare da conoscere. Nell'ordinare i nodi, non guardano lo stato, sono indipendenti dal problema specifico e guardano semplicemente la struttura dell'albero. Non sanno qual è il goal, guardano solo la struttura e guardano quanto è vicino il nodo alla soluzione. **Il Prolog è non informato.**

Le strategie si valutano in base a quattro criteri:

- **Completezza:** la strategia garantisce di trovare una soluzione quando ne esiste una?
- **Complessità temporale:** quanto tempo occorre per trovare una soluzione?
- **Complessità spaziale:** Quanta memoria occorre per effettuare la ricerca?
- **Ottimalità:** la strategia trova la soluzione di "qualità massima" quando ci sono più soluzioni?

4.2 - STRATEGIE DI RICERCA NON INFORMATE

Esistono diversi tipi di strategie:

- **Breadth-first (a costo uniforme)**
- **Depth-first**
- **Depth-first a profondità limitata**
- **Depth-first ad approfondimento iterativo**

```

function GENERAL-SEARCH(problem, strategy) returns a solution, or failure
  initialize the search tree using the initial state of problem
  loop do
    if there are no candidates for expansion then return failure
    choose a leaf node for expansion according to strategy
    if the node contains a goal state then return the corresponding solution
    else expand the node and add the resulting nodes to the search tree
  end

function GENERAL-SEARCH(problem, QUEUING-FN) returns a solution, or failure
  nodes  $\leftarrow$  MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[problem]))
  loop do
    if nodes is empty then return failure
    node  $\leftarrow$  REMOVE-FRONT(nodes)
    if GOAL-TEST[problem] applied to STATE(node) succeeds then return node
    nodes  $\leftarrow$  QUEUING-FN(nodes, EXPAND(node, OPERATORS[problem]))
  end

```

Tramite l'argomento Queuing-Fn viene passata una funzione per accodare i nodi ottenuti dall'espansione.

4.2.1 - BREADTH-FIRST

La profondità del nodo da cui si parte è uguale a 0; la profondità di un qualunque altro nodo è la profondità del genitore più 1. **Questo algoritmo espande sempre i nodi meno profondi dell'albero.** Nel caso peggiore, se abbiamo profondità d e fattore di ramificazione b, il numero massimo di nodi espansi nel caso peggiore sarà b^d (**complessità temporale**) perché $-1 + b + b^2 + b^3 + \dots + (b^d - 1) = b^d$.

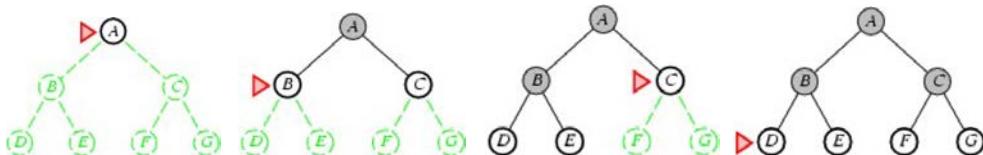
All'ultimo livello sottraiamo 1 perché il goal non viene ulteriormente espanso. **Questo valore coincide anche con la complessità spaziale** (il numero di nodi che manteniamo contemporaneamente aperte più strade).

L'esplorazione dell'albero avviene tenendo contemporaneamente aperte più strade.

Tale strategia garantisce **COMPLETEZZA** ma non permette una efficiente implementazione su sistemi mono-processore.

Trova sempre il cammino a costo minimo se il costo coincide con la profondità.

La strategia a costo uniforme espande sempre il nodo a costo minimo ed è **completa e, a differenza della ricerca in ampiezza, ottimale anche quando gli operatori non hanno costi uniformi** (complessità temporale e spaziale uguale a quella in ampiezza). Essa anziché selezionare in base alla profondità, seleziona in base al fattore costo, quindi accoda i nodi in base al costo ed espande sempre prima quelli a costo più basso. Questa strategia allora è ottima in generale, mentre il breadth first così com'è non è ottimo in generale.



Properties of breadth-first search

Complete?? Yes (if b is finite)

Time?? $1 + b + b^2 + b^3 + \dots + b^d = O(b^d)$, i.e., exponential in d

Space?? $O(b^d)$ (keeps every node in memory)

Optimal?? Yes (if cost = 1 per step); not optimal in general

Space is the big problem; can easily generate nodes at 1MB/sec
so 24hrs = 86GB.

b- massimo fattore di diramazione dell'albero di ricerca

d- profondità della soluzione a costo minimo

m- massima profondità dello spazio degli stati (può essere infinita)

Lo svantaggio principale è l'eccessiva occupazione di memoria.

4.2.2 - DEPTH-FIRST

Questo algoritmo espande per primi nodi più profondi. I nodi di uguale profondità vengono selezionati arbitrariamente (di norma si scelgono quelli più a sinistra). La ricerca in profondità richiede **un'occupazione di memoria molto modesta**. Se si accorge che la strada è sbagliata, scende sull'altra buttando via la strada precedente: **ha sempre e solo una strada attiva all'interno dell'albero**, ha un ramo di preferenza.

Per uno spazio degli stati con fattore di ramificazione b e profondità massima d la ricerca richiede la memorizzazione di b^d nodi. La complessità temporale è invece analoga a quella in ampiezza.

Nel caso peggiore, se abbiamo profondità d e fattore di ramificazione b il numero massimo di nodi espansi nel caso peggiore sarà b^d (**complessità temporale**).

È efficiente dal punto di vista realizzativo: può essere memorizzata una sola strada alla volta (un unico stack). **Può essere non completa con possibili loop in presenza di rami infiniti (come ad esempio l'interprete prolog).**



b- massimo fattore di diramazione dell'albero di ricerca
d- profondità della soluzione a costo minimo
m- massima profondità dello spazio degli stati (può essere infinita)

4.2.3 - RICERCA A PROFONDITÀ LIMITATA

È una variante della **depth-first** in cui si prevede una profondità massima di ricerca. Se non sono sicuro di quanto sia alto l'albero e taglio la profondità, magari la soluzione era in un ramo che abbiamo scartato, semplicemente era dopo il punto di profondità massima. Quando si raggiunge il massimo di profondità o un fallimento si considerano strade alternative della stessa profondità (se esistono), poi minori di una unità e così via (**backtracking**). Evita di scendere lungo rami infiniti. La depth first praticamente costruisce uno stack. Nel modo che abbiamo visto noi, c'è uno stack esplicito in cui mettiamo i dati ed estraiamo i nodi in modo coerente. Qui invece in questa realizzazione c'è una cosa molto furba che è una tecnica molto usata nei programmi di intelligenza artificiale: visto che la ricorsione è implementata sullo stack, uno potrebbe tenere lo **stack implicito** e usare la ricorsione per implementare la strategia.

```

function DEPTH-LIMITED-SEARCH(problem, limit) returns soln/fail/cutoff
    RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)
function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
    cutoff-occurred?  $\leftarrow$  false
    if GOAL-TEST[problem][STATE[node]] then return SOLUTION(node)
    else if DEPTH[node] = limit then return cutoff
    else for each successor in EXPAND(node, problem) do
        result  $\leftarrow$  RECURSIVE-DLS(successor, problem, limit)
        if result = cutoff then cutoff-occurred?  $\leftarrow$  true
        else if result  $\neq$  failure then return result
    if cutoff-occurred? then return cutoff else return failure

```

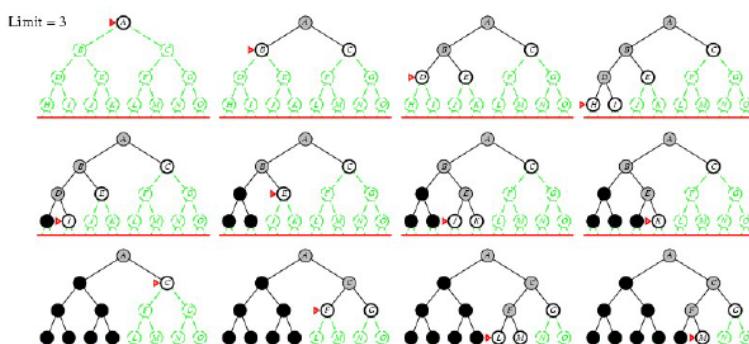
4.2.4 - RICERCA AD APPROFONDIMENTO ITERATIVO o IDS

La ricerca ad approfondimento iterativo evita il problema di scegliere il limite di profondità massimo provando tutti i possibili limiti di profondità: prima 0, poi 1, poi 2 ecc... Combina i vantaggi delle due strategie. È completa e sviluppa un solo ramo alla volta. In realtà tanti stati vengono espansi più volte, ma questo non peggiora sensibilmente i tempi di esecuzione. In particolare, il numero totale di espansioni è: $(d+1)1 + (d)b + (d-1)b^2 + \dots + 3b^{d-2} + 2b^{d-1} + b^d$. In generale è il preferito quando lo spazio di ricerca è molto ampio, può emulare la breadth-first mediante ripetute applicazioni della depth-first con una profondità limite crescente.

```

function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
    inputs: problem, a problem
    for depth  $\leftarrow$  0 to  $\infty$  do
        result  $\leftarrow$  DEPTH-LIMITED-SEARCH(problem, depth)
        if result  $\neq$  cutoff then return result

```



Properties of depth-first search

Complete?? No: fails in infinite-depth spaces, spaces with loops
Modify to avoid repeated states along path
 \Rightarrow complete in finite spaces

Time?? $O(b^m)$: terrible if m is much larger than d
but if solutions are dense, may be much faster than breadth-first

Space?? $O(bm)$, i.e., linear space!

Optimal?? No

Properties of iterative deepening search

Complete?? Yes

Time?? $(d+1)b^0 + db^1 + (d-1)b^2 + \dots + b^d = O(b^d)$

Space?? $O(bd)$

Optimal?? Yes, if step cost = 1

Can be modified to explore uniform-cost tree

b= massimo fattore di diramazione dell'albero di ricerca

d= profondità della soluzione a costo minimo

m= massima profondità dello spazio degli stati (può essere infinita)

4.3 - CONFRONTO FRA LE STRATEGIE DI RICERCA NON INFORMATE

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Time	b^d	b^d	b^m	b^l	b^d	$b^{d/2}$
Space	b^d	b^d	bm	bl	bd	$b^{d/2}$
Optimal?	Yes	Yes	No	No	Yes	Yes
Complete?	Yes	Yes	No	Yes, if $l \geq d$	Yes	Yes

b = fattore di ramificazione;

d = profondità della soluzione;

m=profondità massima dell'albero di ricerca;

l=limite di profondità.

La strategia di ricerca bidirezionale posso immaginarla come una breadth-first perché concettualmente se vado a livelli ho tutti i nodi della frontiera di uno e della frontiera dell'altro e alla fine trovo la soluzione.

Come faccio a battere questi costi? Questi sono i casi peggiori, ma posso usare l'euristica perché è furba e mi aiuta.

4.4 – STRATEGIE DI RICERCA INFORMATE

Un umano ha una capacità di elaborazione non molto elevata, quindi usa strategie furbe.

Es. un giocatore esperto non riesce a scendere di livello nell'albero di ricerca come fa una macchina, però ha una grande capacità euristica che gli consente di tagliare a "colpo d'occhio" delle possibilità: in questo caso intelligenza significa usare bene le nostre risorse.

Newell-Simon:

I metodi di ricerca precedenti in uno spazio di profondità d e fattore di ramificazione b risultano di complessità proporzionale a b^d per trovare un goal in una delle foglie. È inaccettabile per problemi con una certa complessità: serve una conoscenza euristica sul dominio (funzioni di valutazione) per decidere quale nodo espandere per primo.

Le funzioni di valutazione danno una stima computazionale dello sforzo per raggiungere lo stato finale. Questa funzione andrebbe applicata ogni volta che abbiamo un nuovo nodo, non avrebbe senso applicarla in testa (se no sarebbe una breadth first) e neanche tutta in coda.

La funzione di valutazione ci dà una distanza stimata dal goal e in base a quello, lo inseriamo ad una certa distanza dalla coda.

Es. scacchi: se conto i miei pezzi, più ne ho più sono vicino al goal. Nel caso contassi quelli dell'avversario, è il contrario. Contare solo i pezzi però non è un'euristica furba, allora potrei pesarli con il loro valore, poi faccio la differenza con quelli pesati dell'altro. Però certi pezzi sono più o meno importanti a seconda della posizione del pezzo sulla scacchiera e in base alle fasi del gioco, ci sono dei pezzi che sono importanti all'inizio ma che poi alla fine non ci interessano più. Attenzione però, non ci interessa solo la posizione, ma la posizione relativa rispetto agli altri pezzi, e quali pezzi.

Potrei fare molte funzioni euristiche o fare una mega funzione con tantissime variabili da tenere in considerazione: sarebbe molto complicata: una funzione così pesante, se va valutata per ogni nodo, è un casino.

Siccome tutti i problemi che dobbiamo risolvere hanno un tempo limitato, se abbiamo molto tempo dedicato alla funzione di valutazione, scenderemo ad un numero di livelli più limitato.

Non è detto che una funzione di valutazione complicata messa in gara con una più semplice, vinca sempre perché quella più semplice ha qualche livello in più e magari a volte può vincere quella. La differenza è che la funzione di valutazione ben fatta è una stima: ci serve un trade-off.

Infatti, per le strategie informate ci interessano:

- Il tempo speso per valutare mediante una funzione euristica il nodo da espandere. Esso deve corrispondere ad una riduzione nella dimensione dello spazio esplorato.
- Un trade-off tra il tempo necessario per risolvere il problema e il tempo speso per decidere come risolverlo.

Es. gioco del filetto

La distanza dal goal è stimata in base al numero di caselle fuori posto. Potrei trovare anche funzioni euristiche più furbe che non contano solo le posizioni delle tesserine: se una tesserina è praticamente di fianco al posto in cui deve andare è più facile che vada al posto giusto rispetto ad una che è nell'angolo opposto, quindi posso darle un peso. Oppure potrei utilizzare lo stato della casellina vuota.

Le strategie di ricerca informate più note sono:

1. Best First come A* e Greedy
2. Funzioni euristiche

Problemi delle strategie di ricerca informate:

- Non sono ottimali perché possono essere incomplete (stessi difetti della ricerca in profondità)
- Nel caso peggiore, con profondità d e fattore di ramificazione b, il numero massimo di nodi espansi è b^d (complessità temporale)
- Tiene in memoria tutti i nodi, quindi complessità spaziale coincide con quella temporale
- Con una buona funzione euristica, le complessità spaziale e temporale possono essere ridotte

4.4.1 - RICERCA BEST-FIRST

Usa una funzione di valutazione che calcola un numero che rappresenta la desiderabilità relativa all'espansione del nodo. In generale le strategie euristiche che usano una funzione di valutazione che calcola la distanza dal nodo è di tipo Best First (probabilmente best, ovviamente).

Si possono vedere due casi particolari: la ricerca Greedy e la ricerca A* (a star) con le sue declinazioni.

QueuingFn = inserisce i successori in ordine decrescente di desiderabilità.

Algoritmo:

1. Sia L una lista dei nodi iniziali del problema, ordinati secondo la loro distanza dal goal
2. Sia n il primo nodo di L (se L è vuota, fallisce)
3. Se n è il goal, si ferma e ritorna sulla strada percorsa per raggiungerlo
4. Altrimenti rimuove n da L e aggiunge a L tutti i nodi figli di n con label la strada percorsa partendo dal nodo iniziale.
5. Poi ordina L in base alla stima della distanza relativa al goal e ritorna al passo 2.

function BEST-FIRST-SEARCH(*problem*, EVAL-FN) **returns** a solution sequence
inputs: *problem*, a problem
 Eval-Fn, an evaluation function

Queueing-Fn \leftarrow a function that orders nodes by EVAL-FN
return GENERAL-SEARCH(*problem*, *Queueing-Fn*)

La tecnica best-first cerca di trovare il più presto possibile un nodo con distanza 0 dal goal senza curarsi di trovare quel nodo che ha profondità più bassa.

CONVENZIONE: stato iniziale (ordine di espansione)

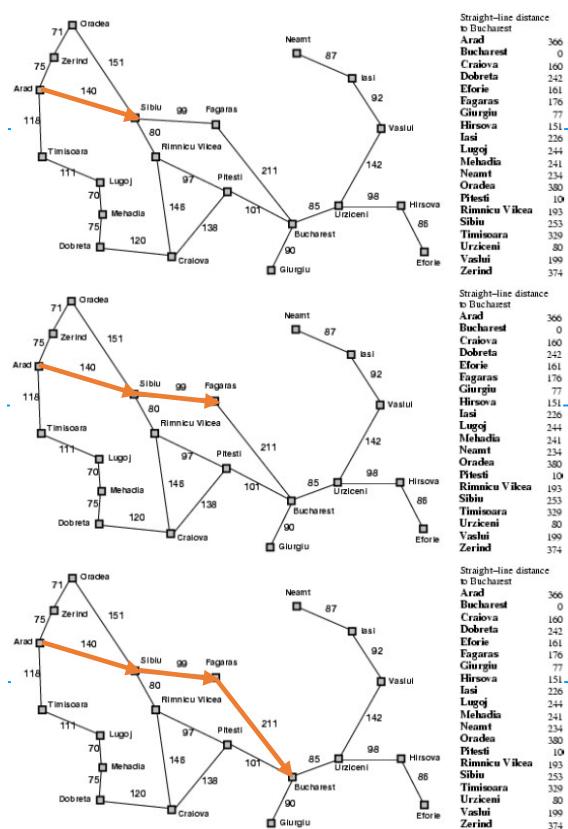
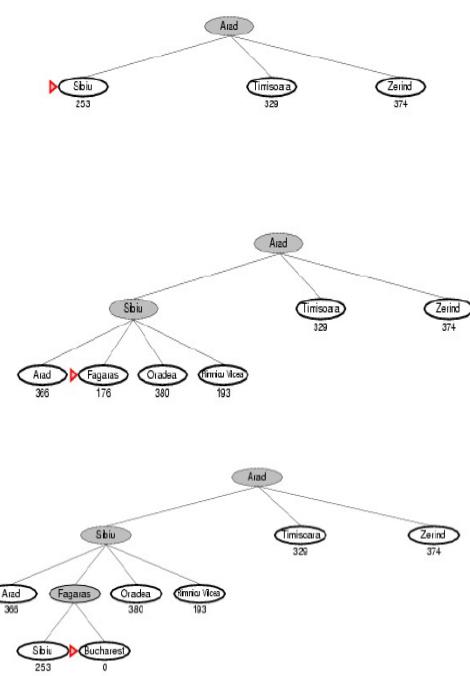
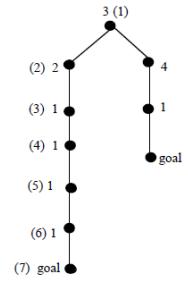
Il nodo radice è il primo ad essere espanso e la sua stima di distanza dal goal è 3.

Il terzo nodo espanso, a sinistra, stima una distanza di 1, ma la funzione di valutazione è evidentemente sbagliata perché non è così. Non è stato espanso il nodo di destra perché la stima di quello di sinistra è 2 ed è minore di 4, ma in realtà avremmo dovuto espandere il nodo di destra.

4.4.2 – GREEDY

La funzione di valutazione è $f(n) = h(n)$ (euristica) e la stima del costo da n al goal è lo stesso. **Espande il nodo che sembra più vicino al nodo.**

Es. Romania con costo in km



La frontiera è formata da tutti i nodi espandibili (bianchi con il contorno nero).

Siccome viene sempre espanso il nodo con costo minimo, alla soluzione si arriva, ma il percorso trovato non è il migliore.

In realtà il costo del cammino è composto da due parti:

- Il costo vero tra lo stato iniziale e il nodo in cui sono arrivato in quel momento
- Stima da quel goal allo stato finale

4.4.3 - A*

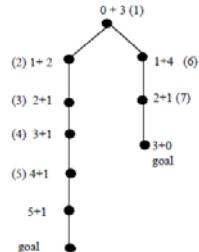
Invece di considerare solo la distanza dal goal, considera anche il costo nel raggiungere il nodo N dalla radice. Si espandono quindi i nodi in ordine crescente di $f(n) = g(n) + h'(n)$ dove $g(n)$ è la profondità del nodo, $h'(n)$ è la distanza dal goal e si sceglie il nodo in cui $f(n)$ è più piccola. Si cerca quindi di combinare i vantaggi della ricerca in profondità e della ricerca a costo uniforme, unendo efficienza, ottimalità e completezza.

$f(n)$ è il costo, che nel primo nodo è 0. Suppongo che il costo per passare da un nodo all'altro sia 1, quindi il nodo di sx diventa 2, ma aggiungo 1 che è il costo per arrivarci.

Quindi espando quello a sx, scendo un po' e quando arrivo al nodo 4+1 a sx, vedo che quello di dx 1+4 ha lo stesso valore: allora espando quello di dx e arrivo al cammino minore.

Ma se il goal fosse dove c'è il (4) o il (5)? **A* non garantisce comunque di trovare cammino migliore.** Guardando l'albero il (6) non viene mai selezionato perché il fatto che dica che la distanza per la soluzione è 4 è pessimista ed è sbagliato (ci sta perché è una stima). Questo significa che si può fare una **stima pessimista o ottimista**.

Nel ramo di sinistra penso sempre di essere ottimista dicendo che manca 1 passo, invece ci arrivo in più passi, a dx invece dico che il goal è a distanza 4, invece ci potrei arrivare in 2.



Algoritmo A*:

1. Sia L una lista dei nodi iniziali del problema
2. Sia n il nodo di L per cui $g(n)+h'(n)$ è minima. Se L è vuota, fallisci.
3. Se n è il goal, si ferma e ritorna sulla strada percorsa per raggiungerlo
4. Altrimenti rimuove n da L e aggiunge a L tutti i nodi figli di n con label la strada percorsa partendo dal nodo iniziale. Ritorna al passo 2.

Si supponga di indicare con $h(n)$ la vera distanza fra il nodo corrente e il goal. La funzione euristica $h'(n)$ è ottimistica se abbiamo sempre che $h'(n) \leq h(n)$ e tale funzione è detta ammissibile.

TEOREMA: Se $h'(n) \leq h(n)$ per ogni nodo, allora l'algoritmo A* troverà sempre il nodo goal ottimale.

L'euristica perfetta $h'=h$ è sempre ammissibile e se $h'=0$ otteniamo sempre una funzione euristica ammissibile perché abbiamo la ricerca breadth-first.

Nella strategia Greedy: $f(n) = h'(n)$.

Nella strategia A*: $f(n) = g(n) + h'(n)$.

Se $f(n) = g(n)$, significa che seleziono sempre i nodi con $g(n)$ più basso, che è una strategia non informata che calcola solo il costo, quindi prende sempre quelli con il costo più basso, e cioè diventa la strategia a costo uniforme.

Se costo per tutti gli altri = 1, la strategia $f(n) = g(n) + h'(n)$ diventa una breadth first perché la profondità è 1, quindi seleziona sempre quelli a profondità migliore.

OTTIMALITÀ DI A*

Si supponga di avere generato un goal sub-ottimo G_2 e di averlo inserito nella coda. Sia n un nodo non espanso nella coda

tale che n è nella strada più breve verso il goal ottimo G.

$f(G_2) = g(G_2)$ poiché $h(G_2) = 0 \Rightarrow g(G_2) > g(G)$ poiché G_2 è sub-ottima

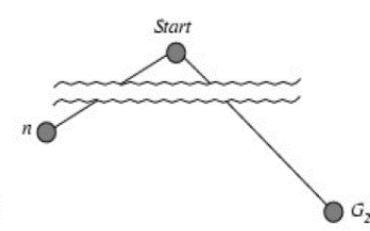
$\Rightarrow f(G) = g(G)$ poiché $h(G) = 0 \Rightarrow f(G_2) > f(G)$ da sopra

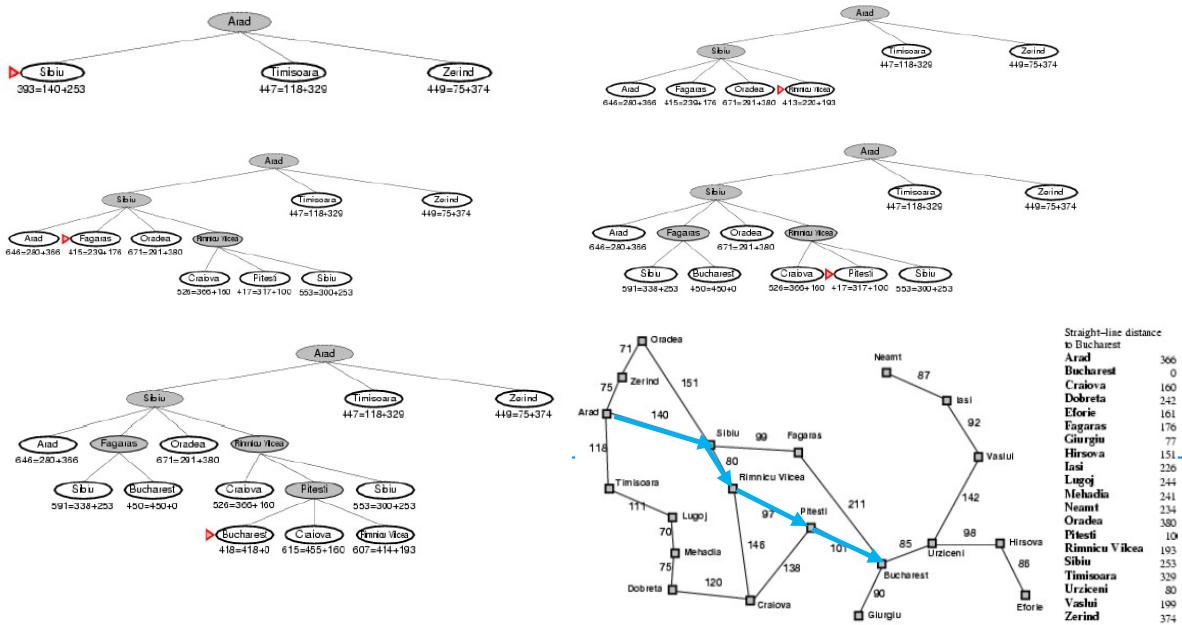
$\Rightarrow h'(n) \leq h(n)$ poiché h' è ammissibile

$\Rightarrow g(n) + h'(n) \leq g(n) + h(n) = f(G)$ (n è sulla strada di G)

$\Rightarrow f(n) \leq f(G)$

$\Rightarrow f(G_2) > f(n)$, e A* non selezionerà mai G_2 per l'espansione.





4.4.4 - FUNZIONI EURISTICHE

Poiché $h' \leq h^* \leq h$, la migliore è h^* . È meglio usare una funzione euristica con valori più alti, a patto che sia ottimista. Spesso il costo di una soluzione esatta per un problema rilassato è una buona euristica per il problema originale. Se la definizione del problema è descritta in un linguaggio formale è possibile costruire problemi rilassati automaticamente. A volte si può utilizzare il massimo fra diverse euristiche: $h'(n) = \max(h_1(n), \dots, h_m(n))$.

DA ALBERI A GRAFI

In questo tipo di problemi si suppone che si possa arrivare alla soluzione in diversi modi, cioè si può raggiungere uno stesso nodo con più strade.

```

function GRAPH-SEARCH( problem, fringe) returns a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
        if STATE[node] is not in closed then
            add STATE[node] to closed
            fringe ← INSERTALL(EXPAND(node, problem), fringe)
    
```

A* PER GRAFI

Due liste:

- Nodi espansi e rimossi dalla lista per evitare di esaminarli nuovamente (**nodi chiusi**);
- Nodi ancora da esaminare (**nodi aperti**).

A* cerca in un grafo invece che in un albero: il grafo può diventare un albero con stati ripetuti, viene aggiunta della lista dei nodi chiusi e si assume che $g(n)$ valuti la distanza minima del nodo n dal nodo di partenza.

Algoritmo A* per i grafì:

1. Sia La una lista aperta dei nodi iniziali del problema
2. Sia n il nodo di La per cui $g(n)+h'(n)$ è minima. Se La è vuota, fallisci.
3. Se n è il goal, si ferma e ritorna sulla strada percorsa per raggiungerlo
4. Altrimenti rimuovi n da La, inseriscilo nella lista dei nodi chiusi Lc e aggiunge a La tutti i nodi figli di n con label la strada percorsa partendo dal nodo iniziale.

- a. Se un nodo figlio è già in L_a , non raggiungerlo, ma aggiornalo con la strada migliore che lo connette al nodo iniziale.
- b. Se un nodo figlio è già in L_c , non aggiungerlo a L_a , ma se il suo costo è migliore, aggiorna il suo costo e i costi dei nodi già espansi che da lui dipendevano.
- c. Ritorna al passo 2.

EURISTICA CONSISTENTE

Una euristica è consistente/monotona se per ogni nodo n e ogni successore n' di n generato da ogni azione a:

- $h(n) = 0$ se lo stato corrispondente coincide con il goal
- $h(n) \leq c(n,a,n') + h(n')$

TEOREMA: Se $h(n)$ è consistente, A^* usando GRAPH-SEARCH è ottimale.

Se h è consistente, abbiamo che $f(n') = f(n)$.

4.4.5 - ALGORITMI COSTRUTTIVI E DI RICERCA LOCALE

Esistono due tipi di algoritmi:

- **Gli algoritmi costruttivi** generano una soluzione ex-novo aggiungendo ad una situazione di partenza (vuota o iniziale) delle componenti in un particolare ordine fino a costruire la soluzione completa.
- **Gli algoritmi di ricerca locale** partono da una soluzione iniziale e iterativamente cercano di rimpiazzare la soluzione corrente con una "migliore" in un intorno della soluzione corrente. In questi casi non interessa la "strada" per raggiungere l'obiettivo.

Diversi problemi (ad esempio, il problema delle 8 regine) hanno la proprietà che la descrizione stessa dello stato contiene tutte le informazioni necessarie per la soluzione (il cammino è irrilevante). In questo caso gli algoritmi con miglioramenti iterativi spesso forniscono l'approccio più pratico.

L'algoritmo dal punto di vista dello pseudo-codice è banale, il problema è lo studio del dominio. Partiamo da una pseudo-soluzione e dobbiamo controllare se quella per noi è davvero la soluzione. Se va bene, finito.

Capire da dove partire in questi algoritmi è fondamentale: l'algoritmo costruisce una struttura detta **struttura dei vicini o neighbourhood**. Da uno stato genera i vicini, quindi ci sarà una funzione che è in grado - data una soluzione - di generare questo spazio (es. i vicini per le 8 regine sono le soluzioni che ottengo scambiando due regine, ma posso farne altri). A questo punto si guarda la soluzione corrente, se non va bene genero i vicini e misuro la **bontà dei vicini**. Se c'è una soluzione migliore di quella in cui sono adesso, allora o questo mi va già bene ed è la soluzione, oppure quella sarà la nuova soluzione di partenza da cui bisognerà generare i nuovi vicini.

Neighbourhood: funzione F che assegna ad ogni soluzione s dell'insieme di soluzioni S un insieme di soluzioni $N(s)$ sottoinsieme di S . Essa è fondamentale e definisce l'insieme delle soluzioni raggiungibili da s in un singolo passo di ricerca. È definita tipicamente attraverso le mosse possibili.

Massimo locale: è una soluzione s t.c. per qualunque $s' \in N(s)$, data una funzione di valutazione f , $f(s) \geq f(s')$. Quando risolviamo un problema di massimizzazione cerchiamo un massimo globale s.t.c. per qualunque s , $f(\text{opt}) > f(s)$. Più è largo il neighbourhood, più è probabile che un massimo locale si annulli e un massimo globale, ma ovviamente aumenta la complessità. (ovviamente per il minimo è l'opposto)

Algoritmo base detto Iterative Improvement (Miglioramento iterativo) parte con una soluzione di tentativo generata in modo casuale o mediante procedimento costruttivo e cerca di migliorare la soluzione corrente muovendosi verso i vicini. Se fra i vicini c'è una soluzione migliore, tale soluzione va a rimpiazzare la corrente, altrimenti termina in un massimo locale.

HILL CLIMBING

Un modo semplice per capire questo algoritmo è di immaginare che tutti gli stati facciano parte di una superficie di un territorio. L'altezza dei punti corrisponde alla loro funzione di valutazione.

Dobbiamo muoverci cercando le sommità più alte tenendo conto solo dello stato corrente e dei vicini immediati.

Hill climbing si muove verso i valori più alti (o bassi) che corrispondono a massimi (o minimi) locali.

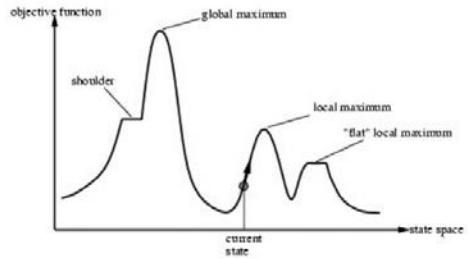
Problema: l'algoritmo può incappare in un minimo (massimo) locale di scarsa qualità: ha il limite di essere un metodo locale che non valuta tutta la situazione, ma solo gli stati abbastanza vicini a quello corrente.

L'algoritmo non gestisce un albero di ricerca, ma solo lo stato e la sua valutazione.

Problemi:

- **Massimi locali:** stati migliori di tutti i vicini, ma peggiori di altri stati che non sono nelle vicinanze. Il metodo ci spinge verso il massimo locale, peggiorando la situazione.
- **Pianori o Altopiani:** zone piatte dello stato di ricerca in cui gli stati vicini hanno tutti lo stesso valore. In quale direzione muoversi (scelta casuale)? Si può estendere il vicinato.
- **Crinali:** zona più alta di quelle adiacenti verso cui dovremmo andare, ma non possiamo andarci in modo diretto. Dobbiamo quindi muoverci in un'altra direzione per raggiungerlo, ad esempio possiamo andare nel punto più basso per poi andare sempre nel punto più alto.

Una possibile soluzione: ripartire con una nuova ricerca da una soluzione di partenza random o generata in modo costruttivo. Si salva poi la soluzione migliore dopo una serie di tentativi.



```

function HILL-CLIMBING(problem) returns a state that is a local maximum
  inputs: problem, a problem
  local variables: current, a node
                  neighbor, a node
  current  $\leftarrow$  MAKE-NODE(INITIAL-STATE[problem])
  loop do
    neighbor  $\leftarrow$  a highest-valued successor of current
    if VALUE[neighbor]  $\leq$  VALUE[current] then return STATE[current]
    current  $\leftarrow$  neighbor
  
```

È come scalare l'Everest in una nebbia fitta con l'amnesia.

Metafora dello scalatore che deve arrivare in cima perché la soluzione è lì: deve andare sempre verso punti che salgono. Però lo scalatore ha vari problemi, per esempio il primo problema dell'algoritmo base è il fatto che non può vedere tutto, ma vede solo i vicini (da qui la "nebbia"). Amnesia perché questo algoritmo non tiene traccia alcuna del percorso e quindi non si ricorda se su un punto ci è già stato (in questa versione base).

Se trovo il massimo valore del vicino e quel valore è più alto rispetto al mio, allora vado in quel punto che diventa il nuovo nodo corrente.

4.4.6 - META EURISTICHE

Insieme di algoritmi, tecniche e studi relativi all'applicazione di criteri euristici per risolvere problemi di ottimizzazione (e quindi di migliorare la ricerca locale con criteri abbastanza generali).

ANT COLONY OPTIMIZATION

Ispirata al comportamento di colonie di insetti. Tali insetti mostrano capacità globali nel trovare, ad esempio, il cammino migliore per arrivare al cibo dal formicaio (algoritmi di ricerca cooperativi). Si usano per la **simulazione**.

TABU SEARCH

La sua caratteristica principale è l'**utilizzo di una memoria per guidare il processo di ricerca** in modo da evitare la **ripetizione** di stati già esplorati.

ALGORITMI GENETICI

Sono **algoritmi evolutivi ispirati ai modelli dell'evoluzione delle specie in natura**. Utilizzano il principio della **selezione naturale** che favorisce gli individui di una popolazione che sono più adatti ad uno specifico ambiente per sopravvivere e riprodursi.

Ogni individuo rappresenta una soluzione con il corrispondente valore della funzione di valutazione (fitness). I tre principali operatori sono: selezione, mutazione e ricombinazione.

SIMULATED ANNEALING

Cerca di evitare il problema dei massimi locali, seguendo in pratica la strategia in salita, ma ogni tanto fa un passo che non porta a un incremento in salita.

Quando rimaniamo bloccati in un massimo locale, invece di cominciare di nuovo casualmente potremmo permettere alla ricerca di fare alcuni passi in discesa per evitare un massimo locale.

Ispirato al processo di raffreddamento dei metalli. In pratica, ci suggerisce di esaminare, ogni tanto, nella ricerca un nodo anche se sembra lontano dalla soluzione

4.4.7 - RICERCA IN GRAFI AND/OR

Abbiamo visto fino ad ora degli algoritmi di ricerca degli stati, ma in realtà in letteratura spesso si parla di alberi and/or. Gli alberi di ricerca che abbiamo visto noi si dovrebbero chiamare tutti alberi OR perché a noi basta trovare la soluzione in uno dei rami. Sono tutte alternative, infatti posso avere un albero di ricerca che ha più soluzioni in cui mi basta trovare una strada, quindi un ramo, per dire che ho trovato la soluzione al mio problema.

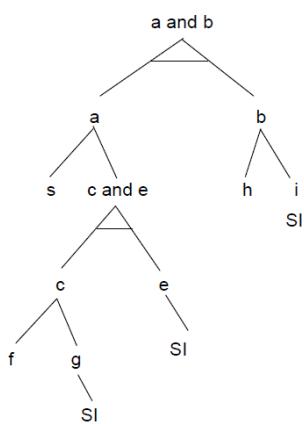
Nell'esempio iniziale del farmaco, c'erano dei rami in AND. Andando in backward (partendo dal goal) ad un certo punto veniva fuori che per verificare la regola, avevo due sotto-goal e avevo fatto un nodo AND perché dovevo verificare tutte e due le condizioni.

Buona parte dell'intelligenza artificiale lavora con regole di questo tipo, quindi bisogna in alcune situazioni utilizzare alberi che hanno logiche di and e logiche di or.

È sempre possibile tradurre un albero AND/OR in un albero OR, quindi tecnicamente potremmo ignorare gli alberi AND/OR. Ha senso perché usando prolog, quello è un sistema logico a regole con le precondizioni in AND (quindi genera alberi concettualmente AND/OR) ma è possibile dimostrare che si può tradurre in un albero OR.

Il grafo (o albero) AND/OR risulta appropriato quando si vogliono rappresentare problemi che si possono risolvere scomponendoli in un insieme di problemi più piccoli che andranno poi tutti risolti.

- g.
- e.
- i.
- s → a.
- c and e → a.
- f → c.
- g → c.
- h → b.
- i → b.
- Goal: a and b



Partendo dal goal (backward), che è la radice, vado a sinistra e devo trovare i rami che fanno match con a, quindi s → a e c&e → a.

Siccome s non è un fatto, è un ramo di fallimento. e è un fatto, quindi quel ramo ha successo.

c non è un fatto e devo trovare regole che matchino con c: → c e g → c.

Siccome g è un fatto, g ha successo, mentre f è un ramo di fallimento.

Il sottoalbero di sinistra quindi è di successo, ora verifico il sottoalbero di destra.

4.4.8 - ALTRE TECNICHE EURISTICHE

- **Scelta in base agli operatori:** regole, task inseriti in un'agenda
- Insita nel controllo e **influenzata IMPLICITAMENTE dall'utente:**
 - **Prolog:** ordine testuale delle regole
 - **OPS:** strategie in base alla forma dell'antecedente (LEX MEA)
- **Influenzata ESPLICATAMENTE dall'utente:** Attribuire alle **regole** dei valori di **priorità** rappresentati come numeri interi; in questo modo l'interprete selezionerà sempre la regola applicabile a più alta priorità.

Es. se ho un problema di diagnostica medica, oltre a dover inserire nel sistema 5/6 mila regole, è impossibile pensare di dover avere una funzione euristica che analizza tutti i casi e che calcola la $h(n)$.

Allora si introducono altre tecniche che guidano la ricerca. Fino ad ora abbiamo sempre parlato dello stato corrente e dello stato finale per vedere quanto è distante dal punto di vista dell'albero, ma potrei lavorare anche su un'altra componente: anziché generare tutti i nodi figli e decidere in base allo stato i figli più promettenti, guardo **quali operatori sono applicabili** oppure **do una certa misura di priorità per ogni operatore**.

Potrei dare priorità per cui parto ad espandere prima il nodo che usa l'operatore prioritario. Questo consente di guidare la ricerca in modo informato e legato al dominio in un modo che spesso è più vicino all'esperto del dominio. Es. sto gestendo una centralina e ci possono essere situazioni di allarme che si generano: se succede questo, deve essere prioritario.

Un sistema è tanto più intelligente quanto ha conoscenza. Inoltre se il sistema applica una regola, mi deve anche dire perché applica una regola e cioè perché ha priorità massima.

Tale approccio presenta però dei **problemi**:

- Risulta particolarmente **complesso individuare il corretto valore di priorità**;
- **L'attribuzione dei valori di priorità è oscura**: diventa dunque **carente** nel sistema la **capacità di spiegazione**.
- **La modificabilità della base di conoscenza diminuisce**: controllo e conoscenza sul dominio non separati; le regole presentano dipendenze implicite.

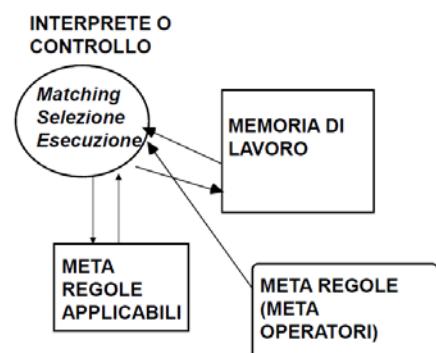
Ci sono allora sistemi che risolvono il problema utilizzando la **Meta-Conoscenza**: essi **ordinano l'agenda** in base a regole del tipo "Usa regole che esprimono situazioni di allarme prima di regole che esprimono situazioni di funzionamento normale" (**MR1**); questa è una **meta-regola euristica sul dominio**, knowledge based. La scelta di quale operatore applicare potrebbe essere il soggetto di un altro knowledge based system che sarà esperto sul dominio e quindi io potrei decidere quale è l'ordinamento migliore per le mie regole di livello oggetto.

VANTAGGI:

- Il **sistema è maggiormente flessibile e modificabile**: un cambiamento nella strategia di controllo implica semplicemente il cambiamento di alcune meta-regole.
- La **strategia di controllo è semplice da capire e descrivere**.
- **Potenti meccanismi di spiegazione** del proprio comportamento

Tereisias è una meta-conoscenza con tre scopi fondamentali:

- Sono utilizzate **meta-regole per definire strategie di controllo**;
- Sono utilizzati **"modelli"** per descrivere **regole di livello oggetto**;
- Sono utilizzati **metodi descrittivi esplicativi** per descrivere la **rappresentazione** utilizzata e dunque poterla modificare



5 – PROBLEMI DI SODDISFACIMENTO DI VINCOLI

5.1 - SODDISFACIMENTO DI VINCOLI

Molti problemi di AI possono essere visti come **problemi di soddisfacimento di vincoli** (CSP o **Constraints Satisfaction Problem**) il cui obiettivo è trovare uno stato del problema che soddisfi un dato insieme di vincoli.

Quando dovremo risolvere dei problemi, spesso andremo in questo ramo, indipendentemente dal campo dell'intelligenza artificiale e ciò permette di riconoscere una classificazione di un problema e di abbattere costi e tempi.

Trovare una soluzione in questo caso significa trovare l'assegnamento delle variabili facendo in modo che tutti i vincoli del problema siano rispettati.

5.1.1 - TIPI DI VINCOLI

Esistono due tipi di vincoli:

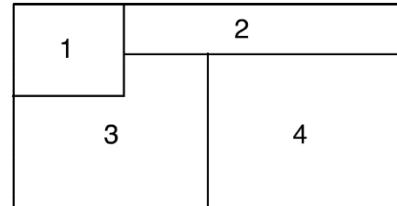
- **Quelli che vincolano i valori delle variabili al dominio suddetto**
Es. Il vincolo $1 \leq x \leq 8$ impone che i valori assunti dalle variabili del problema siano compresi tra i numeri interi 1 e 8: sono vincoli unari.
- **Quelli che devono impedire un attacco reciproco e che impongono, quindi, relazioni tra i valori assunti dalle variabili.**
Es. $x_i \neq x_j$ definisce relazioni tra le variabili e, in particolare, tra due variabili alla volta: sono vincoli binari.

Es. 8 regine

Es. Map Coloring

Supponiamo di dover colorare delle porzioni di un piano, caratterizzate da un numero, in modo tale che due regioni contigue siano colorate da colori diversi. Supponiamo anche di aver a disposizione i colori red (r), green (g) e blu (b).

Variabili: regioni. Domini: colori permessi. Vincoli: Regioni adiacenti devono avere colori diversi.



5.2 - TIPI DI CSP

Esistono due tipi di CSP e variano in funzione del tipo delle variabili che trattano:

- **Variabili discrete**
 - **Domini finiti**
 - n variabili, con dimensione d (quelli che vedremo).
 - Interi, stringhe, ecc...
 - Es. Job-scheduling: variabili rappresentano giorni di inizio-fine per ogni lavoro.
 - Vincoli di durata $\text{StartJob}_1 + 5 \leq \text{StartJob}_2$
- **Variabili continue**
 - Ricerca operativa

Formalmente un **CSP** può essere **definito su un insieme finito di variabili**: (X_1, X_2, \dots, X_n) i cui valori appartengono a **domini finiti di definizione** (D_1, D_2, \dots, D_n), e su un insieme di vincoli. Un **vincolo** $c(X_{i1}, X_{i2}, \dots, X_{ik})$ tra k variabili è un **sottoinsieme del prodotto cartesiano** $D_{i1} \times D_{i2} \times \dots \times D_{ik}$ che specifica quali valori delle variabili sono compatibili con le altre. Tale sottoinsieme non deve essere **definito esplicitamente** ma è **rappresentato in termini di relazioni**. Una soluzione ad un CSP prevede un assegnamento di tutte le variabili che soddisfino tutti i vincoli.

In un CSP sono fondamentali:

- **State** è definito da variabili X_i con valori presi dai domini D_i . Qui le variabili non hanno un valore, sono unbound
- **Goal test** è un insieme di vincoli che specificano le combinazioni di valori permessi (in modo intensionale). Se operatori sono assegnamenti di valori a variabili, il mio albero nelle foglie avrà tutte le variabili assegnate. Il goal test è qualcosa che implementerò e che controlla che quello stato finale rispetta i vincoli: se li rispetta è una soluzione, se no non lo è.
- **Operatori** possono essere assegnamenti di valori a variabili (labeling)

L'algoritmo CSP utilizza:

- **Stato iniziale:** assegnamento vuoto { }
- **Funzione Successore:** assegna un valore ad una variabile non ancora legata (in modo che sia legale con gli assegnamenti già fatti). Fallisce se non esiste
- **Goal test:** l'assegnamento è completo (tutte le variabili sono legate).

Caratteristiche:

- Schema identico per tutti i CSP
- Profondità limitata a n se n sono le variabili → usa depth-first search
- La strada è irrilevante.
- Problema commutativo con d^n foglie (se d è la cardinalità dei domini)

5.2.1 - ALBERO DECISIONALE

Un possibile albero decisionale per un CSP si ottiene, dopo aver stabilito un ordinamento per le variabili, facendo corrispondere ad ogni livello dell'albero l'assegnamento di una variabile e ad ogni nodo la scelta di un possibile valore da dare alla variabile corrispondente al livello del nodo stesso.

Ogni foglia dell'albero rappresenta quindi un assegnamento di valori a tutte le variabili. Se tale assegnamento soddisfa tutti i vincoli, allora la foglia corrispondente rappresenta una soluzione del problema, altrimenti rappresenta un fallimento.

La ricerca di una soluzione è equivalente all'esplorazione dell'albero decisionale per trovare una foglia-soluzione. In un problema di n variabili in cui i domini hanno tutti la stessa cardinalità d, il numero di foglie di un albero decisionale così costruito è pari a d^n .

Esempio. Se ho un problema di 10 variabili in cui ogni dominio ha cardinalità 10, esistono 10 miliardi di foglie.

5.3 - APPROCCI RISOLUZIONE CSP

Esistono due possibili approcci per la risoluzione di un CSP dato:

- **Basata sulle tecniche di consistenza:** essi sono basati sulla propagazione dei vincoli per derivare un problema più semplice di quello (completo) originale. Tipicamente si applicano per primi.
- **Basata su algoritmi di propagazione:** essi sono basati sulla propagazione dei vincoli per eliminare a priori, durante la ricerca, porzioni dell'albero decisionale che porterebbero ad un sicuro fallimento (compatibilmente con le scelte già effettuate). Tipicamente si applicano dopo le tecniche di consistenza.

Senza perdita di generalità, ci riferiremo da ora in poi a CSP su vincoli binari.

Consideriamo una ricerca depth-first in un albero decisionale. Si tende a scendere di livello nell'albero fino a quando o si sono assegnate tutte le variabili, e quindi si è trovata una soluzione, oppure non è più possibile trovare un valore (la sequenza corrente non può portare a una soluzione ammissibile); quindi si esegue un'altra scelta sull'ultima variabile della sequenza stessa. L'algoritmo ha tre gradi di libertà:

- **La scelta nell'ordinamento delle variabili;** → euristica
- **La scelta nell'ordine di selezione del valore da attribuire alla variabile corrente;** → euristica
- **La propagazione effettuata in ciascun nodo.** → differenzia le diverse strategie:
 - **Algoritmi senza propagazione:** Generate and Test, Standard Backtracking
 - **Algoritmi di Propagazione:** Forward Checking, (Partial and Full) Look Ahead

5.4 - ALGORITMI GENERATIVI/SENZA PROPAGAZIONE

Gli algoritmi generativi sono detti anche algoritmi senza propagazione e le due tecniche che usano i vincoli a posteriori sono:

- Il **Generate and Test (GT)**
- Lo **Standard Backtracking (SB)**



5.4.1 – GENERATE AND TEST

L'interprete del linguaggio sviluppa e visita un albero decisionale percorrendolo in profondità assegnando valori alle variabili senza preoccuparsi di verificare la consistenza con gli altri vincoli.

Es. Map Coloring: albero decisionale con 4⁵ foglie

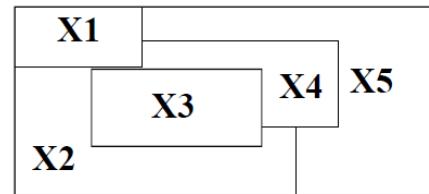
Vincoli di dominio: X1, X2, X3, X4, X5:: [rosso, giallo, verde, blu]

Vincoli topologici:

$X_1 \neq X_2, X_1 \neq X_3, X_1 \neq X_4, X_1 \neq X_5,$

$X_2 \neq X_3, X_2 \neq X_4, X_2 \neq X_5,$

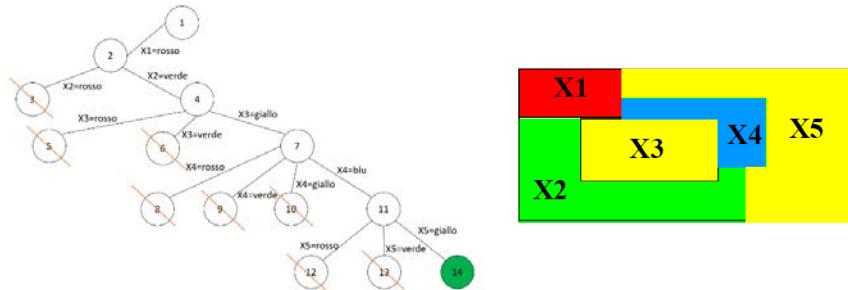
$X_3 \neq X_4, X_3 \neq X_5$



Solo in un secondo tempo questa tecnica considera gli altri vincoli rifiutando la soluzione trovata perché incompatibile con i vincoli del problema. A questo punto inizia la procedura di backtracking tentando con la seconda permutazione e così via finché non si trova una soluzione. **Inefficienza di base: I vincoli sono utilizzati per limitare lo spazio delle soluzioni dopo che la ricerca è stata effettuata**, quindi a posteriori. Il numero delle possibili permutazioni aumenta con il fattoriale del numero di termini da permutare.

5.4.2 – STANDARD BACKTRACKING

Sebbene migliore della precedente anche questa tecnica prevede un **utilizzo a posteriori dei vincoli**: a ogni istanziazione di una variabile si preoccupa di verificare la coerenza della variabile appena istanziata con quelle assegnate precedentemente. Quindi l'utilizzo dei vincoli è più efficace del precedente perché non si prosegue la ricerca in rami che, ai primi livelli dell'albero, presentano delle contraddizioni.



La Depth-first search per i CSP con singolo assegnamento di variabili è chiamata standard backtracking: SB search è l'algoritmo di ricerca non-informata basilare per CSP.

I vincoli sono utilizzati all'indietro (backward) e portano a una effettiva riduzione dello spazio di ricerca. Tuttavia questa riduzione viene fatta a posteriori, cioè dopo aver effettuato il tentativo.

```

function BACKTRACKING-SEARCH( csp ) returns a solution, or failure
    return RECURSIVE-BACKTRACKING( { }, csp )
function RECURSIVE-BACKTRACKING( assignment, csp ) returns a solution, or failure
    if assignment is complete then return assignment
    var  $\leftarrow$  SELECT-UNASSIGNED-VARIABLE( Variables/csp, assignment, csp )
    for each value in ORDER-DOMAIN-VALUES( var, assignment, csp ) do
        if value is consistent with assignment according to Constraints[csp] then
            add { var = value } to assignment
            result  $\leftarrow$  RECURSIVE-BACKTRACKING( assignment, csp )
            if result  $\neq$  failure then return result
            remove { var = value } from assignment
    return failure
  
```

5.5 - ALGORITMI DI PROPAGAZIONE

L'idea che sta alla base degli algoritmi di propagazione a priori consiste in un **utilizzo attivo dei vincoli** nella guida della computazione e nel cosiddetto pruning a priori dell'albero decisionale associando, a ciascuna variabile, l'**insieme di valori ammissibili rimanenti dopo ogni assegnazione**. Questi insiemi (domini) vengono perciò ridotti nel corso della computazione permettendo di scegliere per le variabili ancora libere valori ammissibili con le variabili già istanziate senza più considerare i vincoli che le legano.

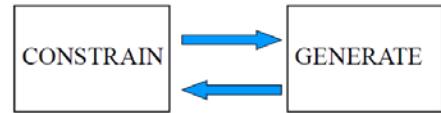
Se una variabile, che deve avere un valore, non ce l'ha perché ha dominio vuoto, allora quella soluzione non va bene.

Le tecniche di propagazione sono metodi di ricerca più intelligenti che tentano di prevenire i fallimenti anziché recuperare fallimenti già avvenuti, appunto facendo **Pruning a priori dell'albero delle decisioni**.

Queste tecniche utilizzano le relazioni tra le variabili del problema, i vincoli, per ridurre lo spazio di ricerca prima di arrivare al fallimento. Vengono così **eliminati rami dell'albero che porterebbero ad un sicuro insuccesso** limitando inutili backtracking.

Le tecniche utilizzate sono:

- Il Forward Checking (FC)
- Il Looking Ahead (LA)
 - Partial LA
 - Full LA



5.5.1 – FORWARD CHECKING

Viene utilizzata, dopo ogni assegnamento, la propagazione dei vincoli che consiste nell'eliminazione dei valori incompatibili con quello appena istanziato dai domini delle variabili non ancora istanziate. Questo metodo si rivela **molto efficace soprattutto quando le ultime variabili ancora libere sono associate ad un insieme di valori ammissibili ridotto e perciò risultano molto vincolate e facilmente assegnabili**.

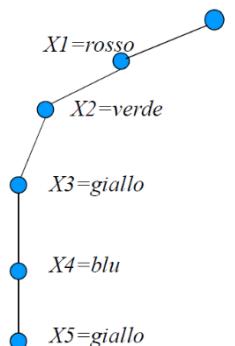
Se il dominio associato ad una variabile libera presenta un solo valore l'assegnamento può essere effettuato senza sforzo computazionale. Se ad un certo punto della computazione ci si accorge che un dominio associato ad una variabile risulta vuoto il meccanismo del Forward Checking fallisce senza proseguire in tentativi e backtracking.

L'assegnazione di un valore ad una variabile ha ripercussioni sull'insieme dei valori disponibili per le variabili ancora libere. In questo modo i vincoli agiscono in avanti (forward) e limitano lo spazio delle soluzioni prima che vengano effettuati tentativi su di esso. Questo metodo è uno dei più usati.

Es. Immaginiamo di avere 4 variabili e abbiamo già assegnato X_1 e X_2 , sicuri che i valori tra di loro sono compatibili perché abbiamo fatto prima Standard Backtracking. Siamo al labeling di X_2 : stiamo per assegnarle un valore.

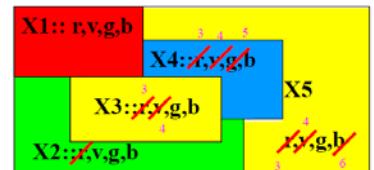
1. Controllo i vincoli tra x_2 e x_1 con Standard Backtracking
2. Se va bene, con il Forward Checking controllo che il vincolo tra x_2 e x_3 sia compatibile
3. Lo faccio anche per x_4 , quindi controllo dominio tra x_2 e x_4

In pratica controllo che tutte le variabili successive contengano valori compatibili con quelli che voglio assegnare alla variabile in cui sono io.



Es. map coloring

Al punto 3, prima di passare a $X_2=\text{verde}$, guardo tutti i valori che sono compatibili con X_1 : dal dominio di TUTTE le altre variabili, devo cancellare il rosso. Al punto 4, stessa cosa per il verde: devo cancellare da tutte le altre variabili il verde. E così via.

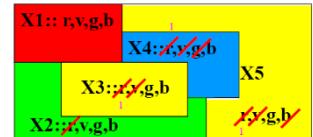


5.5.2 – LOOKING AHEAD

Questo metodo è il più completo per quel che riguarda il pruning a priori dell'albero decisionale. Ad ogni istanziazione viene controllata, come per il Forward Checking, la compatibilità dei vincoli contenenti la variabile appena assegnata con le precedenti (istanziate) e le successive (libere). In più viene sviluppato il **look ahead (sguardo in avanti)** che controlla l'esistenza, nei domini associati alle variabili ancora libere, di valori compatibili con i vincoli contenenti solo variabili non istanziate. I domini associati a ogni variabile vengono ridotti propagando anche le relazioni contenenti coppie di variabili non istanziate. Viene verificata quindi la possibilità di una futura assegnazione consistente fra coppie delle variabili libere.

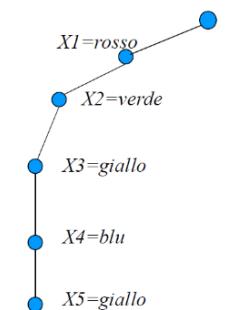
In pratica considero e controllo due variabili, entrambe ancora non legate, e i loro domini. Guardo dentro la variabile x_k , prendo i valori dei suoi domini, guardo la variabile x_{k+1} e a questo punto controllo che esista nella variabile successiva almeno un valore per cui quella variabile e l'altra variabile rispettino i vincoli. Io so quali sono i domini, perciò se per ogni valore che c'è nel dominio esiste almeno un dominio in quello di x_{k+1} che soddisfa il vincolo, allora potrei considerare

quella strada, se invece non trovo nessun valore che rispetta il vincolo, evito quel percorso e, visto che non ho ancora assegnato nulla, cancello il valore dalla variabile x_k .



PARTIAL LOOK AHEAD (PLA)

Si ha una propagazione dei vincoli contenenti la variabile X_h (non ancora istanziata) e le variabili "future", ossia le variabili X_{h+1}, \dots, X_n . Per ogni variabile non ancora assegnata X_{k+1}, \dots, X_n , deve esistere un valore per il quale sia possibile trovare, per tutte le altre variabili "successive" non ancora assegnate, almeno un valore compatibile con esso.



Es. map coloring

PLA rispetto a FC non aumenta il pruning dell'albero in questo caso, semplicemente anticipa di un livello il fallimento. Prima di assegnare X_2 , al passo 1, devo controllare in tutti i domini se ci sono soluzioni, quindi ad esempio il giallo va bene e blu va bene, quindi anche verde per x_2 va bene: allora assegno verde. Quando assegno verde però dobbiamo fare anche Forward Checking, quindi il dominio di x_3 diventa vuoto, allora il dominio di x_4 ci accorgiamo che non è compatibile: ci accorgiamo prima che è una soluzione di fallimento.

FULL LOOK AHEAD (FLA)

FLA: se V_k è il valore appena assegnato alla variabile X_k , si ha una propagazione dei vincoli contenenti la variabile X_h (non ancora istanziata) e tutte le variabili non ancora assegnate, ossia le variabili $X_{k+1}, \dots, X_{h-1}, X_{h+1}, \dots, X_n$.

Per ogni variabile non ancora assegnata X_{k+1}, \dots, X_n deve esistere un valore per il quale sia possibile trovare, per tutte le variabili non ancora assegnate, almeno un valore compatibile con esso.

Questo è computazionalmente più costoso, ma l'albero sarà più potato perché in pratica non controllo solo le variabili successive, ma anche le variabili precedenti.

Es. $X_0 < X_1 < X_2 < X_3$ con domini per $X_1, X_2, X_3 :: [1,2,3]$



PLA, verifica:

- Per ogni valore in D_1 se esiste almeno un valore in D_2 e almeno un valore in D_3 compatibili (sono eliminati i valori di D_1 per i quali non esiste alcun valore compatibile in D_2 o in D_3)
- Per ogni valore in D_2 se esiste almeno un valore in D_3 compatibile (sono eliminati i valori di D_2 per i quali non esiste alcun valore compatibile in D_3)

FLA verifica le stesse condizioni di PLA e inoltre controlla:

- Per ogni valore in D_2 se esiste almeno un valore in D_1 compatibile (sono eliminati i valori di D_2 per i quali non esiste alcun valore compatibile in D_1)
- Per ogni valore in D_3 se esiste almeno un valore in D_2 e almeno un valore in D_1 compatibili (sono eliminati i valori di D_3 per i quali non esiste alcun valore compatibile in D_1 o in D_2)

PLA $\rightarrow X_1 :: [1,2], X_2 :: [1,2], X_3 :: [1,2,3]$. FLA $\rightarrow X_1 :: [1,2], X_2 :: [2], X_3 :: [3]$

5.6 – CLASSIFICAZIONE DELLE EURISTICHE

La scelta dell'ordinamento delle variabili e la scelta dell'ordine di selezione dei valori rimangono a disposizione del programmatore. Le euristiche potranno agire quindi su questi due gradi di libertà per cercare di garantire il raggiungimento di una buona soluzione in tempi ragionevoli anche per i problemi più complessi.

Le euristiche possono essere classificate in:

- **Euristiche per la selezione della variabile:** determinano quale deve essere la prossima variabile da istanziare. Le due euristiche più comunemente usate sono il **first-fail** (o **MRV: Minimum Remaining Values**) che sceglie la variabile con il dominio di cardinalità minore, e il **most-constrained principle** che sceglie la variabile legata a più vincoli. Entrambe queste euristiche decidono di istanziare prima le variabili più difficili da assegnare.
- **Euristiche per la selezione del valore:** determinano quale valore assegnare alla variabile selezionata. Si segue in genere il principio di scegliere prima il valore che si ritiene abbia più probabilità di successo (**least constraining principle**).

Un'ulteriore classificazione è la seguente:

- **Euristiche statiche:** determinano l'ordine in cui le variabili (o i valori) vengono scelti prima di iniziare la ricerca; tale ordine rimane invariato durante tutta la ricerca.
- **Euristiche dinamiche:** scelgono la prossima selezione da effettuare ogni volta che una nuova selezione viene richiesta (quindi ad ogni passo di labeling). Le euristiche dinamiche sono **potenzialmente migliori** (meno backtracking).

La determinazione dell'euristica perfetta (che non richiede backtracking) è un problema che ha, in genere, la stessa complessità del problema originale. Bisognerà quindi trovare un compromesso.

5.7 – TECNICHE DI CONSISTENZA

Differenza fondamentale: al contrario degli algoritmi di propagazione che propagano i vincoli in seguito a istanziazioni delle variabili coinvolte nel problema, **le tecniche di consistenza riducono il problema originale eliminando dai domini delle variabili i valori che non possono comparire in una soluzione finale**.

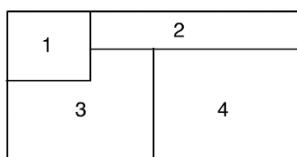
Possono essere **applicate staticamente oppure ad ogni passo di assegnamento** (labeling) come potenti tecniche di propagazione per le variabili non ancora istanziate.

Tutte le tecniche di consistenza sono basate su una rappresentazione del problema come una rete (grafo) di vincoli. Gli archi possono essere **orientati o non orientati**: ad esempio il vincolo $>$ viene rappresentato da un **arco orientato**, mentre il vincolo \neq da un **arco semplice** (non orientato o doppiamente orientato).

5.7.1 - CONSTRAINT GRAPH

Per ogni CSP esiste un grafo (**constraint graph**) in cui i nodi rappresentano le variabili e gli archi i vincoli tra le variabili costituenti i nodi del grafo:

- I **vincoli binari (R)** collegano due nodi X_i e X_j
- I **vincoli unari** sono rappresentati da archi che iniziano e terminano sullo stesso nodo X_i



Es. map coloring: colori red (r), green (g) e blu (b).

Il constraint-graph corrispondente è il seguente. Tuttavia, esistono combinazioni di valori non compatibili tra loro (es: $X_1=r$, $X_2=r$, $X_3=r$, $X_4=r$). Esistono diversi algoritmi che realizzano gradi diversi di consistenza.

5.7.2 – GRADI DI CONSISTENZA

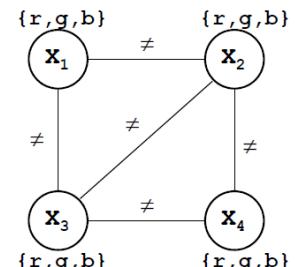
Questi algoritmi non devono cancellare qualcosa che fa parte della soluzione, se no sono incompleti. Essi provano a cancellare qualcosa che sicuramente non farà parte della soluzione. È chiaro che se mi trovo alla fine un dominio vuoto a forza di cancellare, allora quel grafo non ha soluzione.

Ciò è molto interessante perché **spesso ci si trova davanti a situazioni che sono ad esempio troppo vincolate: a questo punto allora o si rilassano dei vincoli (cioè si tolgono) oppure si modifica il problema aggiungendo più variabili o più valori nei domini**.

Es. devo assegnare dei task alle risorse e non ci riesco, allora aumento le risorse, oppure diminuisco i task oppure aumento la deadline (anziché un giorno, ne metto due).

In letteratura si chiamano tecniche di consistenza gli algoritmi che fanno quindi questa cosa e tali tecniche hanno un **grado crescente di consistenza**, cioè ci sono algoritmi che vanno a fare sempre più verifiche cancellando sempre più valori dai domini.

Queste tecniche non necessariamente arrivano ad avere un dominio in cui ci sono solo valori che fanno parte della soluzione, a volte alcuni valori non fanno parte della soluzione. **Più aumenta il grado di consistenza, più arrivo alla situazione in cui nel dominio ho solo valori che sono nella soluzione, ma è così costoso che generalmente ci si ferma qualche grado prima**.



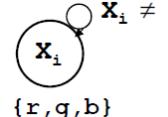
NODE CONSISTENCY – grado 1

NODE-CONSISTENCY: consistenza di grado 1

- Un nodo di un grafo di vincoli è consistente se per ogni valore $X_i \in D_i$ il vincolo unario su X_i è soddisfatto.

Nell'esempio il nodo non è node consistent perché il valore $g \in D_i$ viola il vincolo unario $P(i)$ su X_i . Per rendere il nodo consistente è necessario eliminare dal dominio di X_i il valore g . **Un grafo è node consistente se tutti i suoi nodi sono consistenti.**

Questo è il **grado più semplice perché coinvolge una sola variabile**. X_i può valere r, g, b . Il vincolo unario è il vincolo del variabile sul suo dominio, es. x_i deve essere \neq da g .



In questo caso se questo è il mio grafo, non è node consistent perché per la definizione non è vero che per ogni valore del suo dominio, quel vincolo unario è soddisfatto. **La tecnica di consistenza, cerca di rendere questo grafo node consistent cancellando dal dominio il valore g .**

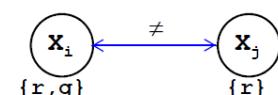
ARC CONSISTENCY – grado 2

- ARC CONSISTENCY: un arco $A(i,j)$ è consistente se per ogni valore $X \in D_i$ esiste almeno un valore $Y \in D_j$ tale che il vincolo tra i e j $P(i,j)$ sia soddisfatto

Quando faccio questa consistenza, immagino di aver già fatto la node consistency perché se no potremmo ricadere in un assurdo, quindi supponiamo di partire da un grafo che è già node consistent.

Ora coinvolgiamo due nodi del grafo: andiamo a vedere se questi archi sono consistenti: se ho un arco che va da i a j (se è biunivoco ovviamente ci sarà anche quello che va da j a i), questo è consistente se per ogni valore di x che appartiene al dominio i , esiste almeno un valore in j tale che il vincolo tra i e j sia soddisfatto.

Se ho due nodi e per un valore del primo non esiste almeno un valore nel secondo dominio, quell'elemento non farà parte di una soluzione.

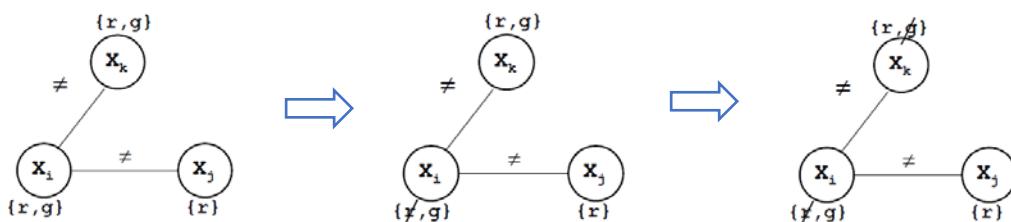


Quello nell'esempio non è arc consistente: devo cancellare r perché non farà parte della soluzione in quanto non è diverso da r . Ovviamente poi devo fare la stessa cosa in senso inverso perché dobbiamo immaginare che l'arco sia bidirezionale (le frecce blu le ho disegnate io).

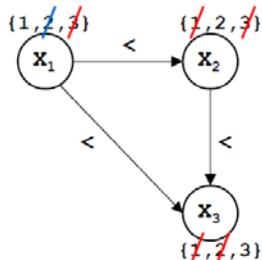
PROCEDIMENTO ITERATIVO

L'algoritmo di arc consistenza non termina facendo un solo giro. Potrebbe succedere, cancellando dei valori, che un arco che era consistente, ad un certo punto non lo sia più. L'algoritmo terminerà quando il grafo arriverà ad una forma in cui non cancello più nulla da nessun arco, devo quindi fare un **Procedimento Iterativo** che poi mi farà arrivare ad un punto fisso in cui lo **stato non cambia più**, detto **quiescenza**.

In questo grafo abbiamo x_k , x_i e x_j e questo grafo non è arc consistent. Al primo passo dell'algoritmo avrei un grafo che ha in x_k r, g , in x_i g e in x_j r , ma in realtà facendo solo un giro non ho ottenuto un grafo arc consistent, devo ricominciare. Magari non rifaccio tutti i controlli, ma faccio una coda di archi da controllare in cui metto gli archi in base ai quali c'è stata una riduzione.



Es. $x_1 < x_2 < x_3 \rightarrow x_1 > x_2, x_1 < x_3, x_2 < x_3$ con dominii $X_1, X_2, X_3 :: [1, 2, 3]$



Nella prima iterazione (rossa), parto da x_1 . Per ogni valore di x_1 , devo guardare i valori di x_2 tali che il primo sia minore del secondo, quindi con 1, trovo 2 e 3, con 2 trovo 3 e per 3 non trovo nessuno, quindi cancello 3. Ma non ho finito qua, metto il nodo nella coda di quelli da ricontrillare.

Siccome tutti i domini sono stati ridotti, faccio una seconda iterazione (blu): ora in x_1 cancello 2 che prima non era stato cancellato.

Faccio infine una terza iterazione in cui trovo che il primo arco è soddisfatto, il secondo anche e il terzo pure. Non cancellando nulla, allora sono arrivato alla quiescenza, cioè non ho modificato domini e quindi gli elementi che sono all'interno del dominio (in questo caso singoli) sono validi.

Il controllo della consistenza dell'arco può essere applicato sia prima della ricerca, come pre-processing per produrre un problema semplificato, oppure come passo di propagazione (in analogia al Look Ahead) dopo ogni assegnamento di variabile: è spesso chiamato **Maintaining Arc Consistency (MAC)**. In questo caso l'**algoritmo completo** per il controllo di consistenza di un arco è chiamato **AC-3** ed utilizza una coda di archi (**queue**), ciclando finché non è vuota. Non appena il dominio di una variabile X_i è ridotto, si ri-aggiungono a queue gli archi (X_k, X_i) per ciascuna variabile X_k collegata da un arco incidente su X_i .

```

function AC-3(csp) returns the CSP, possibly with reduced domains
  inputs: csp, a binary CSP with variables  $\{X_1, X_2, \dots, X_n\}$ 
  local variables: queue, a queue of arcs, initially all the arcs in csp
  while queue is not empty do
     $(X_i, X_j) \leftarrow \text{REMOVE-FIRST}(\text{queue})$ 
    if RM-INCONSISTENT-VALUES( $X_i, X_j$ ) then
      for each  $X_k$  in NEIGHBORS[ $X_i$ ] do
        add  $(X_k, X_i)$  to queue

function RM-INCONSISTENT-VALUES( $X_i, X_j$ ) returns true iff remove a value
  removed  $\leftarrow$  false
  for each  $x$  in DOMAIN[ $X_i$ ] do
    if no value  $y$  in DOMAIN[ $X_j$ ] allows  $(x, y)$  to satisfy constraint( $X_i, X_j$ )
      then delete  $x$  from DOMAIN[ $X_i$ ]; removed  $\leftarrow$  true
  return removed

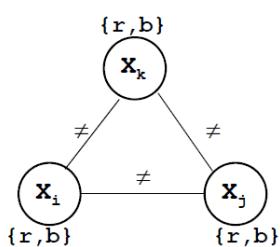
```

PATH CONSISTENCY – grado 3

PATH CONSISTENCY: Un cammino tra i nodi (i, j, k) è path consistente se, per ogni valore $x \in D_i$, e $y \in D_j$ (che rispettano la node e la arc-consistenza) esiste un valore $z \in D_k$ che soddisfa i vincoli $P(i, k)$ e $P(j, k)$.

La consistenza di grado 3 si ottiene partendo da un grafo arc consistente. La consistenza del vincolo unario $P(k)$ è garantita dalla node consistenza della rete.

Es. supponiamo di considerare la rete di vincoli relativa a una istanza del map coloring problem: questa rete è arc-consistente, infatti, per ogni valore di ciascun dominio, esiste almeno un valore in ogni altro dominio che soddisfa il vincolo esistente tra i due nodi. Tuttavia, è immediato verificare che la rete non presenta soluzioni quindi non è path-consistente: presi i valori $r \in D_i$ e $b \in D_k$, non esiste nessun valore appartenente a D_j che soddisfi contemporaneamente i vincoli $P(i, j)$ e $P(j, k)$.



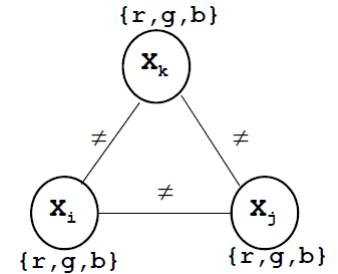
Strutture dati aggiuntive, per ciascuna coppia di variabili (per ciascun arco) si memorizzano le coppie di valori compatibili per i quali esiste nel dominio della terza variabile un valore compatibile (esempio: per l'arco da X_i a X_j , nessuna delle coppie $\langle r, b \rangle$ e $\langle b, r \rangle$ ha un valore a supporto nel dominio di X_k). **Verificare, per questo esempio, che la rete non è path consistent equivale alla situazione di fallimento nella ricerca.**

Aggiungendo un colore alla paletta disponibile: r,g,b, si ha che la rete è Path Consistent!

Quindi ora verificare la consistenza di grado $n=3$ per questa rete che ha esattamente $n=3$ variabili equivale a verificare che esiste una

soluzione (Nota bene: poiché il grado di consistenza è identico al numero di variabili).

Come risultato teorico quindi possiamo dire che se abbiamo N nodi e N è anche il grado della consistenza, c'è sicuramente una soluzione, è però troppo costoso quindi tipicamente si tende a fare una arc consistency con un interleaving tra il labeling e l'arc consistency.



TEOREMA: Se ogni cammino di lunghezza 2 di un grafo completo è path-consistente,
allora l'intera rete è path-consistente.

Un constraint-graph non completo può essere reso completo aggiungendo, tra le variabili non vincolate, vincoli completamente rilassati: sempre veri.

K-CONSISTENZA

Scelti valori per ogni k-1-PLA (Partial Look Ahead) di variabili consistenti con i vincoli imposti dal problema, si cerca un valore per ogni k-esima variabile che soddisfa i vincoli tra tutte le k variabili. Se tale valore esiste allora le k variabili sono k consistenti. In generale, se un grafo contenente n variabili è k-consistente con $k < n$, allora per trovare una soluzione è necessaria una ricerca nello spazio restante.

Se un grafo contenente n variabili è n-consistente, allora si può trovare una soluzione senza ricerca: suppongo di avere un grafo con due nodi: x_1 e x_2 entrambi con "ab" e sull'arco ho un diverso. Faccio la node consistency e non cancello nulla. Faccio l'arc consistency e c'è un teorema che dice che c'è una soluzione perché ho due nodi e due valori. Ho due soluzioni: ho sicuramente una soluzione con $x_1=a$ e con $x_2=b$. Questo significa che non ho bisogno di fare alcuna ricerca, ma attenzione, non significa che presi a casaccio dei valori allora ho una soluzione, non è inteso in questo modo il "senza ricerca".

Freuder nel 1978 ha definito un algoritmo generale per rendere una rete k-consistente con k qualunque. Tuttavia, rendere una rete di vincoli contenente n variabili n-consistente ha una complessità esponenziale in n (costa quanto effettuare la ricerca sul problema originale).

5.7.3 - CONSTRAINT SOLVER

I più diffusi constraint solver fanno uso delle tecniche viste fino ad ora. **In Constraint Programming tipicamente si usa un algoritmo di labeling con le euristiche viste ed arc-consistency per la propagazione di vincoli su variabili non istanziate.**

I vincoli sono visti come **componenti software che incapsulano un algoritmo di filtering**. Molto spesso l'**algoritmo di filtering** che effettua la propagazione **non è general purpose** come quelli visti finora, ma si basa sulla semantica del vincolo per motivi di efficienza.

Una caratteristica fondamentale dei constraint solver è la presenza di vincoli n-ari, anche detti vincoli globali. Chiaramente per raggiungere la consistenza di un vincolo n-ario (**Generalized Arc Consistency**), in linea di principio, si dovrebbe applicare la n-consistenza, che ha **complessità esponenziale**. Tuttavia esistono vincoli particolari per cui raggiungere la n-consistenza ha **costo polinomiale**. Per gli altri ci si accontenta di un'**approssimazione della Generalized Arc Consistency**.

Es. $X::[1..10]$, $Y::[1..10]$, $X>Y$.

È inutile che io mi metta a guardare valore per valore se c'è arc consistency, perché ci metterei sessant'anni. Sapendo le proprietà di questi domini e cioè che ho un range di numeri da 1 a 10, allora mi basta prendere il limite inferiore e quello superiore. Questo vincolo è AC (Arc-Consistent) se $\min(X) > \min(Y)$ e se $\max(X) < \max(Y)$.

5.8 - CSP E PROBLEMI DI OTTIMIZZAZIONE

La maggior parte dei problemi CSP sono **NP-difficili**, cioè sono problemi per i quali non è ancora stato trovato, e probabilmente non esiste, un algoritmo in grado di trovare la soluzione in un tempo polinomiale nella dimensione del problema. Un **Constraint Optimization Problem (COP)** è un **CSP** o Problema di Soddisfacimento di Vincoli **in cui viene aggiunto un obiettivo**.

Un COP è quindi formalmente descrivibile come un CSP il cui scopo non è solo trovare una soluzione ammissibile, ma la **soluzione ottima secondo un certo criterio di valutazione**.

Dato un algoritmo generale in grado di risolvere qualsiasi CSP si può utilizzare tale algoritmo per risolvere anche qualsiasi COP. Infatti, dopo aver descritto il problema in termini di variabili, domini e vincoli, **basta aggiungere una variabile ulteriore che rappresenta la funzione obiettivo, il costo**. Ogni volta che si trova una soluzione al CSP viene aggiunto un nuovo vincolo che garantisce che ogni soluzione futura avrà un valore della funzione obiettivo migliore. Questo procedimento continua finché non sarà più possibile trovare alcuna soluzione. **L'ultima soluzione trovata è la soluzione ottima**.

5.9 – TECNICHE ALTERNATIVE

Una tecnica alternativa è la **Programmazione Lineare** che utilizza tecniche numeriche per risolvere sistemi di equazioni (o disequazioni) lineari + ottimizzazione (Es. algoritmo del simplex). Lavora molto bene per vincoli lineari, mentre non lavora molto bene per problemi discreti. In questi casi si usa **Constraint Processing**, anche per problemi di vincoli non su dati numerici.

6 – GIOCHI

L'IA si applica anche nell'ambito dei giochi, in particolare si parla di:

- **Ambiente multi-agente** che deve tenere conto della presenza di un "avversario": qui cambia che io non solo devo cercare una soluzione, ma devo tenere conto che anche il mio avversario sta facendo una ricerca, quindi il mio modo di agire è influenzato.
- **Teoria dei giochi**, che è una branca dell'economia
- **Giochi formali** (hanno regole quindi sono più facili da usare in certi contesti)
Attualmente le macchine hanno superato gli esseri umani in molti giochi quali Othello, Dama, Scacchi, Backgammon e GO

L'intelligenza artificiale considera giochi con le seguenti proprietà:

- **Sono giochi a due giocatori (min e max) in cui le mosse sono alternate e le funzioni di utilità sono complementari (vince e perde);**
- **Sono giochi con conoscenza perfetta/completa in cui i giocatori hanno la stessa informazione.** Tutti vediamo tutto: la scacchiera la vediamo tutti e due e lo stato del gioco è visibile da tutti. Stiamo escludendo ad esempio quasi tutti i giochi con le carte perché io non so cos'ha in mano il mio avversario.

Lo svolgersi del gioco si può interpretare come un albero in cui la radice è la posizione di partenza e le foglie le posizioni finali.

6.1 – ALGORITMO MIN-MAX

L'algoritmo min-max è progettato per determinare la strategia ottimale per "Max" e per suggerirgli, di conseguenza, la prima mossa migliore da compiere; per fare questo, ipotizza che "Min" faccia la scelta a lui più favorevole. Non è interessante la "strada", ma solo la prossima mossa.

Io a partire da a, quale strada faccio? Posso aver fatto l'albero più bello del mondo, ma serve un modo per capire tra quelle tre scelte qual è la migliore. Prima arrivo alle foglie e le etichetto, poi a partire dalle foglie faccio un algoritmo di propagazione all'indietro che etichetta i genitori fino ad arrivare alla radice.

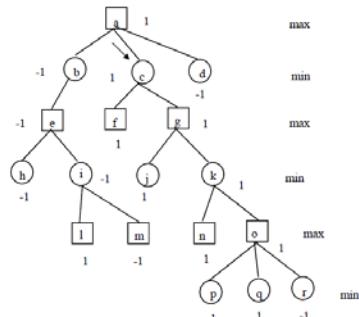
Quando ho +1, vince il Max. Quando ho -1, vince il Min. Quando ho 0, patta. Per i quadri tocca muovere al max, per i cerchi al min.

Consideriamo il nodo o, al penultimo livello: deve muovere il max. Il gioco termina comunque. Se muove "p" ed "r" perde, se muove "q" vince. Supponiamo che muova q → o diventa una posizione vincente. (+1).

Ora consideriamo il nodo k, al terzultimo livello: comunque muova min, perderà. Quindi l'etichetta è (+1) perché è a favore di max.

Consideriamo ora il nodo i, sempre al terzultimo livello: min ha un'opzione vincente dunque (-1).

Allora un nodo con max che deve muovere ha come label il massimo delle labels dei figli. Viceversa per min.



I giocatori li abbiamo chiamati max e min e hanno operazioni complementari. Uno dei due vuole ottenere il massimo dei valori, l'altro vuole ottenere il minimo dei valori (perché la sua funzione è complementare). A partire dallo stato corrente bisogna individuare le mosse da fare per andare su una determinata strada, poi partirà l'altro player (c'è un'alternanza simmetrica nel gioco).

Se pensiamo ad un albero che viene sviluppato completamente fino alle foglie l'algoritmo ha due passi:

1. Sviluppa l'albero in cui non ci sono costi
2. Da lì devo risalire alla mossa che devo fare dallo stato corrente

L'algoritmo così pensato partirebbe da nodi che hanno figli etichettati, quindi prima dai nodi padri delle foglie. Per max si può etichettare anche nodo genitore con il valore massimo e per min etichettare con il valore minimo.

Questo ci consentirà di etichettare i padri delle foglie in modo ricorsivo con un algoritmo all'indietro fino ad etichettare il nodo iniziale della situazione corrente. A quel punto avendo etichettato le possibili mosse dallo stato corrente, sceglierò quell'azione che mi porta al valore massimo/minimo in base se gioca max/min.

Questo purtroppo però tipicamente non si può fare, sia per il costo, sia perché di solito c'è un tempo limite per ogni giocatore, non può metterci un'eternità.

Per valutare un nodo n:

- 1) Espandi l'intero albero sotto n;
- 2) Valuta le foglie come vincenti per max o min;
- 3) Seleziona un nodo n' senza etichetta i cui figli sono etichettati. Se non esiste ritorna il valore assegnato ad n;
- 4) Se n' è un nodo in cui deve muovere min assegna ad esso il valore minimo dei figli, se deve muovere max assegna il valore massimo dei figli. Ritorna a 3.

Complessità in tempo e spazio = b^d .

Per valutare un nodo n in un albero di gioco:

- 1) Metti in L = (n) i nodi non ancora espansi.
 - 2) Sia x il primo nodo in L. Se x = n e c'è un valore assegnato a esso ritorna questo valore.
 - 3) Altrimenti se x ha un valore assegnato Vx, sia p il padre di x e Vp il valore provvisorio a esso assegnato.
 - 4) Se p è un nodo min, Vp = min(Vp, Vx), altrimenti Vp = max(Vp, Vx). Rimuovi x da L e torna allo step 2.
 - 5) Se ad x non è assegnato alcun valore ed è un nodo terminale, assegnagli 0, -1, o 0. Lascia x in L perchè si dovranno aggiornare gli antenati e ritorna al passo 2.
- Se a x non è stato assegnato un valore e non è un nodo terminale, assegna a Vx = -infinito se X è un max e Vx = + infinito se è un min. Aggiungi i figli di x a L **in testa** e ritorna allo step 2.

Complessità in spazio: bd.

6.1.1 – PROPRIETÀ DI MIN-MAX

- **Completo, se l'albero è finito**
- **Ottimale, se l'avversario gioca al meglio**
- **Complessità temporale $O(b^m)$**
- **Complessità spaziale $O(bm)$ (depth-first)**

Per gli scacchi, che hanno $b \approx 35$ e $m \approx 100$ per partite "ragionevoli", è impensabile tale soluzione! Dobbiamo potare l'albero.

6.2 – ALGORITMO MIN-MAX RIVISTO

Se devo sviluppare tutto l'albero la procedura è molto inefficiente perché è esponenziale. Se b è il fattore di ramificazione e d sono i livelli allora il numero dei nodi diventa b^d . La soluzione proposta da Shannon nel 1949 dice: **si guarda avanti solo per un po' e si valutano le mosse fino ad un nodo non terminale ritenuto di successo**. In pratica si applica min-max fino ad una certa profondità. In questo modo si utilizza una certa **funzione di valutazione per stimare la bontà di un certo nodo**.

- $e(n) = -1$ sicuramente vincente per min;
- $e(n) = +1$ sicuramente vincente per max;
- $e(n) = 0$ circa le stesse probabilità;

Poi valori intermedi per $e(n)$.

Per valutare un nodo n in un albero di gioco:

1. Metti in L = (n) i nodi non ancora espansi.
2. Sia x il primo nodo in L. Se x = n e c'è un valore assegnato a esso ritorna questo valore.
3. Altrimenti se x ha un valore assegnato Vx, sia p il padre di x e Vp il valore provvisorio a esso assegnato.
 - a. Se p è un nodo min, Vp = min(Vp, Vx)
 - b. Altrimenti Vp = max(Vp, Vx).

Rimuovi x da L e torna allo step 2.

4. Se ad x non è assegnato alcun valore ed è un nodo terminale, oppure decidiamo di non espandere l'albero ulteriormente, assegna il valore utilizzando la funzione di valutazione $e(x)$. Lascia x in L perché si dovranno aggiornare gli antenati e ritorna al passo 2.
5. Aggiornare gli antenati e ritorna al passo 2.
6. Se a x non è stato assegnato un valore e non è un nodo terminale, assegna a $V_x = -\infty$ se X è un max e $V_x = +\infty$ se è un min. Aggiungi i figli di X a L e ritorna allo step 2.

```

function MINIMAX-DECISION(state) returns an action
  v  $\leftarrow$  MAX-VALUE(state)
  return the action in SUCCESSORS(state) with value v

function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v  $\leftarrow -\infty$ 
  for a, s in SUCCESSORS(state) do
    v  $\leftarrow \text{MAX}(v, \text{MIN-VALUE}(s))$ 
  return v

function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v  $\leftarrow \infty$ 
  for a, s in SUCCESSORS(state) do
    v  $\leftarrow \text{MIN}(v, \text{MAX-VALUE}(s))$ 
  return v

```

Se $e(n)$ fosse perfetta non avrei problemi: espanderei solo i figli della radice per decidere cosa fare. Purtroppo non lo è, quindi una soluzione possibile e semplice anche dal punto di vista computazionale è: espando sempre fino ad una certa profondità p.

Problemi:

- Mosse tatticamente più complicate (con valori che si modificano più ampiamente per $e(n)$) dovrebbero essere valutate con più profondità fino alla quiescenza (valori di $e(n)$ che cambiano molto lentamente).
- Effetto orizzonte: Con mosse non particolarmente utili, allungo la profondità dell'albero di ricerca oltre p, se p è la profondità massima, per cui le mosse essenziali non vengono in realtà prese in considerazione.

Soluzione: a volte conviene fare una ricerca secondaria, mirata sulla mossa scelta.

6.3 – TAGLI ALFA BETA

I computer che giocano semplicemente cercano in alberi secondo certe proprietà matematiche, perciò considerano anche mosse e nodi che non si verificheranno mai. Si deve cercare di ridurre lo spazio di ricerca e la tecnica più conosciuta è quella del taglio alfa-beta.

Se io sono un nodo max e ho già raggiunto il nodo max possibile, perché devo sviluppare anche le altre possibilità? Analogamente vale per il nodo min: se sono un nodo min e ho già trovato un -1, quindi è la situazione in cui vinco, perché devo cercare anche tutte le altre soluzioni?

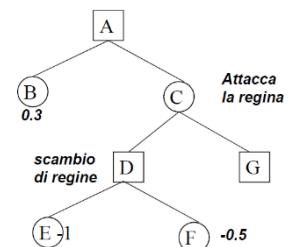
Si consideri un nodo N nell'albero. Il giocatore si muoverà verso quel nodo? Se il giocatore ha una scelta migliore M a livello del nodo genitore od in un qualunque punto di scelta precedente, N non sarà mai selezionato. Se raggiungiamo questa conclusione possiamo eliminare N.

Sia Alfa il valore della scelta migliore trovata sulla strada di MAX (il più alto) e Beta il valore della scelta migliore trovata sulla strada di MIN (il più basso), l'algoritmo aggiorna Alfa e Beta e taglia quando trova valori peggiori.

Es. Immagino che A sia un nodo max e che abbia i due figli B e C.

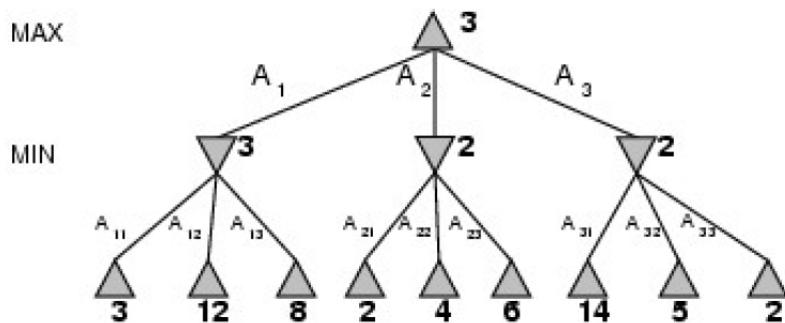
Suppongo per B che abbia già ottenuto in qualche modo un valore che è 0.3 (sono valori euristici). Immaginiamo che abbia sviluppato il sottoalbero di C.

D ha due possibilità negative e visto che D è un massimizzatore e C è un minimizzatore, da D otterrò -0.5 (è la sua possibilità migliore).

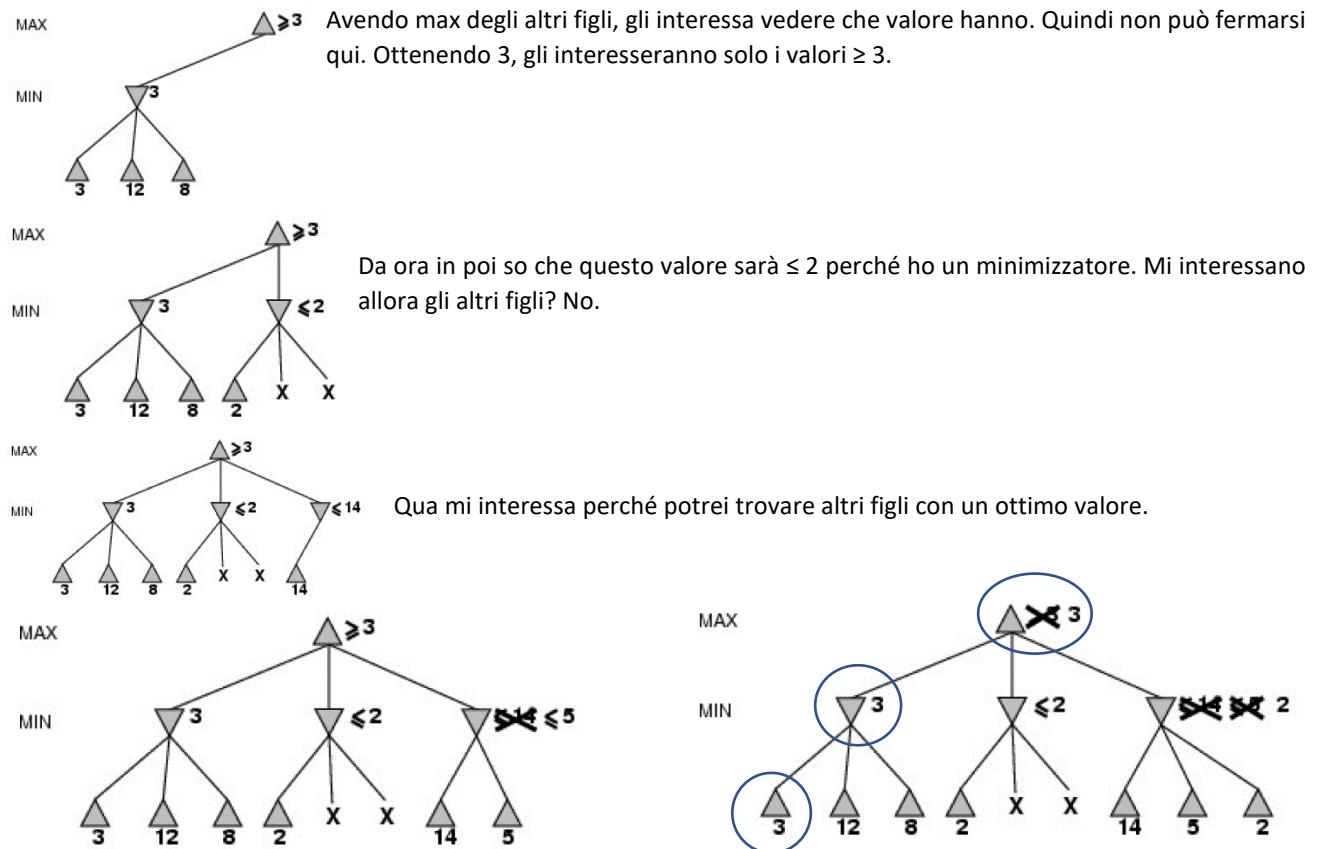


C'è un minimizzatore, quindi se G è ancora nella lista dei nodi da sviluppare (infatti non ha etichetta), sotto di lui potrebbe esserci il mondo, però G è -0.5, quindi posso scegliere tutti i nodi del mondo sotto G, ma avrò sempre e solo valori ≤ -0.5 . Allora a questo punto A, che è un massimizzatore, cosa potrà ottenere sviluppando ulteriormente? Gli interesseranno valori almeno uguali a 0.3, quindi otterrà valori che siano ≥ 0.3 . Avendo supposto che di là troverò solo valori ≤ -0.5 , allora a cosa mi serve andare in quella strada e sviluppare i nodi? Rimango in B, tanto so che di meglio non posso ottenere.

Es. MIN MAX



Es. ALFA BETA



Principi:

- Si genera l'albero depth-first, left-to-right
- Si propagano i valori (stimati) a partire dalle foglie
- I (temporanei) valori nei MAX-nodes sono ALPHA-values
- I (temporanei) valori nei MIN-nodes sono BETA-values
- Se un ALPHA-value è \geq di un Beta-value di un nodo discendente: stop generazione figli discendente.
- Se un Beta-value \leq ad un Alpha-value di un nodo discendente: stop generazione figli discendente.

6.4 – ALGORITMO ALFA-BETA

Per valutare un nodo n in un albero di gioco:

1. Metti in L = {n} i nodi non ancora espansi.
2. Sia x il primo nodo in L. Se x = n e c'è un valore assegnato a esso ritorna questo valore.
3. Altrimenti se x ha un valore assegnato Vx, sia p il padre di x. Se a x non è assegnato un valore vai al passo 5.
 - a. Determiniamo se p ed i suoi figli possono essere eliminati dall'albero. **Se p è un nodo min, sia alfa il massimo di tutti i correnti valori assegnati ai fratelli di p e dei nodi min che sono antenati di p.**
 - b. Se non ci sono questi valori alfa = -infinito.
 - c. Se Vx ≤ alfa rimuovi p e tutti i suoi discendenti da L (dualmente se p è un max).
4. Se p non può essere eliminato, sia Vp il suo valore corrente. Se p è un nodo min, Vp = min(Vp, Vx), altrimenti Vp = max(Vp, Vx). Rimuovi x da L e torna allo step 2.
5. Se a x non è assegnato alcun valore ed è un nodo terminale, oppure decidiamo di non espandere l'albero ulteriormente, assegna il valore utilizzando la funzione di valutazione e(x). Lascia x in L perché si dovranno aggiornare gli antenati e ritorna al passo 2.
6. Se a x non è stato assegnato un valore e non è un nodo terminale, assegna a Vx = -infinito se X è un max e Vx = +infinito se è un min. Aggiungi i figli di X ad L e ritorna allo step 2.

Il valore che corrisponde ad alfa per gli antenati max è chiamato beta.

1. Determiniamo se p ed i suoi figli possono essere eliminati dall'albero. Se p è un nodo max, sia beta il massimo di tutti i correnti valori assegnati ai fratelli di p e dei nodi max che sono antenati di p.
2. Se non ci sono questi valori beta = + infinito.
3. Se Vx ≥ beta rimuovi p e tutti i suoi discendenti da L

```

function ALPHA-BETA-SEARCH(state) returns an action
  inputs: state, current state in game
  v ← MAX-VALUE(state, -∞, +∞)
  return the action in SUCCESSORS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
  inputs: state, current state in game
          α, the value of the best alternative for MAX along the path to state
          β, the value of the best alternative for MIN along the path to state
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← -∞
  for a, s in SUCCESSORS(state) do
    v ← MAX(v, MIN-VALUE(s, α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v

function MIN-VALUE(state, α, β) returns a utility value
  inputs: state, current state in game
          α, the value of the best alternative for MAX along the path to state
          β, the value of the best alternative for MIN along the path to state
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for a, s in SUCCESSORS(state) do
    v ← MIN(v, MAX-VALUE(s, α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v

```

6.4.1 - EFFICACIA DEI TAGLI

Worst case: se valutiamo sempre i nodi peggiori, i nodi valutati successivamente risultano sempre nella linea corrente di ricerca e non c'è nessun taglio.

Best case: quando i nodi migliori sono valutati per primi. I restanti vengono sempre tagliati. In questo caso si va a ridurre il numero dei nodi da b^d a circa $b^{d/2}$. In pratica, si riduce della radice quadrata il fattore di ramificazione, ovvero si può

guardare due volte più avanti nello stesso tempo. Nel caso medio con distribuzione casuale dei valori ai nodi, il numero di nodi diventa circa $b^{3d/4}$, quindi è importante ordinare bene i figli di un nodo.

Si noti inoltre che tutti i risultati qui citati sono per un albero di gioco "ideale" con profondità e ramificazione fissate per tutti i rami e nodi.

Gli attuali programmi scendono circa di 7 livelli ed elaborano circa 250.000 posizioni per volta ma in particolari condizioni possono arrivare fino a 20 livelli e 700.000 posizioni.

Inoltre quasi tutti i programmi utilizzano il tempo che il giocatore umano impiega per scegliere la sua mossa per esplorare altre strade. **Il giocatore umano, in realtà sembra non scenda mai per più di 5 livelli, e con tagli notevoli.** Non utilizza poi una funzione di valutazione definita in modo metodico (usa il "colpo d'occhio").

Il computer non è in grado di adottare una strategia globale, ma questa limitazione è spesso compensata da **non commettere sviste o dimenticanze**.

Tutti i programmi di scacchi, inoltre, **consultano la libreria delle aperture** (ci sono un centinaio di aperture ormai completamente esplorate e che possono condizionare tutta la partita). Mentre il computer è fortissimo nel centro partita, **il giocatore umano è più abile nel finale**, dove la strategia posizionale è meno importante.

Ma oggi i programmi di scacchi, proprio per ovviare a questo inconveniente, tendono a utilizzare librerie ed algoritmi specializzati per il finale.

Es. Deepblue → ha battuto in un match di sei partite il campione del mondo di scacchi

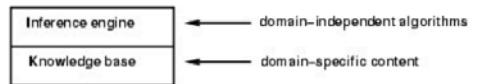
I giocatori artificiali possono classificarsi in due categorie in base al fatto che utilizzino hardware generico (PC) o hardware speciale (è il caso di Deep Blue). In particolare, Deep Blue utilizza una macchina parallela general-purpose a 32 processori più 512 chip specializzati per la generazione di mosse e valutazione.

Altri approcci: deep learning e metodi subsimbolici.

7 – INTRODUZIONE ALLA LOGICA

Concetti base della logica:

- **Sintassi:** struttura formale delle sentenze
- **Semantica:** verità di sentenze rispetto ad interpretazioni/modelli
- **Conseguenza logica (entailment):** sentenza necessariamente vera data un'altra sentenza
- **Inferenza:** derivare (sintatticamente) sentenze da altre sentenze
- **Correttezza (soundness):** la derivazione produce solo sentenze che sono conseguenza logica
- **Completezza (completeness):** la derivazione può produrre tutte le conseguenze logiche



La **Knowledge base (KB)** o **Base di Conoscenza** è intesa come l'insieme di sentenze scritte in un linguaggio formale. Noi considereremo come linguaggio formale la logica dei predici del primo ordine.

Inference Engine: strutture dati ed algoritmi per manipolare la KB ed arrivare ad una risposta.

La logica è quella scienza che fornisce all'uomo gli strumenti indispensabili per controllare con sicurezza la rigorosità dei ragionamenti. La logica fornisce gli strumenti formali per:

- **Analizzare le inferenze** in termini di operazioni su espressioni simboliche;
- **Dedurre conseguenze** da certe premesse;
- **Studiare la verità o falsità di certe proposizioni** data la verità o falsità di altre proposizioni;
- **Stabilire la consistenza e la validità di una data teoria.**

L'approccio della logica è molto rigido ma di fatto non sappiamo se l'essere umano basa tutti i suoi ragionamenti sulla logica. C'è anche qualcos'altro che non sappiamo cos'è.

Se uno fa un ragionamento logico, tipicamente è riconosciuto e comunemente accettato da altri, ma questo non significa che possiamo risolvere tutti i problemi del mondo con la logica.

La logica lavora su espressioni e frasi di tipo simbolico; non fa dei calcoli, quindi non usa numeri, ed è in grado di fare dei tipi di ragionamento (elencati all'inizio nell'introduzione: induttivo, deduttivo, analogico). Può studiare la verità/falsità di alcune proposizioni e fondamentalmente è uno strumento molto potente.

La logica non è una novità in un corso di ingegneria informatica perché noi usiamo dei **calcolatori basati sulla logica booleana**: questo è importante perché la logica è molto usata, più di quello che pensiamo.

La logica è utilizzata:

- **In IA come linguaggio formale per la rappresentazione di conoscenza**
 - Semantica non ambigua
 - Sistemi formali di inferenza
- **Per sistemi di dimostrazione automatica di teoremi** e studio di meccanismi efficienti per la dimostrazione
- **Per la progettazione di reti logiche**
- **Nei database relazionali**, come potente **linguaggio per l'interrogazione intelligente**
- **Come linguaggio di specifica** di programmi che per eseguire hanno bisogno di prove formali di correttezza
- **Come un vero e proprio linguaggio di programmazione** (programmazione logica e PROLOG)

Ci sono tanti tipi di logica, noi vedremo quella **classica** che si suddivide in **due classi principali**:

- **Logica proposizionale** (non tratta variabili, es. reti logiche)
- **Logica dei predici**

Questi due tipi di logica permettono di **esprimere proposizioni** (cioè frasi) e **relazioni tra proposizioni**.

La principale **differenza** tra le due classi è in **termini di espressività**: **nella logica dei predici è possibile esprimere variabili e quantificazioni, mentre questo non è possibile nella logica proposizionale**.

7.1 - LOGICA PREDICATI PRIMO ORDINE

Il linguaggio della logica dei predici del primo ordine è definito da:

- Una **sintassi**: caratteristiche strutturali del linguaggio formale (mediante una **grammatica**) senza attribuire alcun significato ai simboli
- Una **semantica**, che **interpreta** le **frasi sintatticamente corrette** del linguaggio. Si dà una interpretazione alle formule stabilendo se una frase è vera o falsa

7.1.1 - ALFABETO

Consiste di cinque insiemi:

- **C**: Insieme dei simboli di **costante**
- **F**: Insieme dei simboli di **funzione**
- **P**: Insieme dei simboli di **predicato** (o relazione)
- **V**: Insieme dei simboli di **variabile**
- **I connettivi logici**:
 - ~ (negazione),
 - ^ (congiunzione o and),
 - v (disgiunzione o or),
 - → (implicazione),
 - ↔ (equivalenza),
 - le parentesi ()
 - e i quantificatori esistenziale (**Ǝ**) e universale (**∀**).

1.1.2 - SINTASSI (Prolog)

- **Costanti**: singole entità del dominio del discorso. (Es. "maria", "giovanna", "3" con **iniziale minuscola**)
- **Variabili**: entità non note del dominio (Es. X, Y con **iniziale maiuscola**)
- **Funzioni n-arie**: individua univocamente un oggetto del dominio del discorso mediante una relazione tra altri "n" oggetti del dominio. (Es. madre(maria))
- **Importante: le funzioni, in logica, non presuppongono alcun concetto di valutazione**
- **Predicati n-ari**: generica relazione (che può essere **vera o falsa**) fra "n" oggetti del dominio del discorso (Es. parente(giovanna,maria))

Una delle tante differenze è che la funzione madre(maria) concettualmente ci fa un mapping con un'entità che sarà la madre di maria, ma nella logica non è valutata. **madre(maria) denota già la madre di maria in senso simbolico e strutturale, non viene valutata per poi arrivare a giovanna.** È un po' diverso da come siamo abituati a vedere le funzioni nei linguaggi di programmazione.

In questo sistema Prolog like, la sintassi delle funzioni è uguale a quella dei predici.

Termine (definito ricorsivamente):

- una **variabile** è un termine
- una **costante** è un termine
- se f è un simbolo di funzione n-aria e t₁, ..., t_n sono termini, allora **f(t₁, ..., t_n)** è un termine
- (Es. maria,f(X))
- Atomo o formula atomica: l'applicazione di un simbolo di predicato n-ario p a n termini t₁, ..., t_n: p(t₁, ..., t_n). (Es. parente(giovanna,maria))

Espressione o formula: sequenza di simboli appartenenti all'alfabeto.

parente(giovanna, maria)	(E1)	Questo è un predicato.
Ǝ X (uomo(X) ^ felice(X))	(E2)	Esiste almeno un uomo che è felice.
∀ X (uomo(X) → mortale(X))	(E3)	Sillogismo aristotelico: ogni uomo è mortale
Ǝ X (uomo(X) ^	(E4)	
Ǝ X (uomo(f(X)	(E5)	

Formule ben formate (fbf): frasi sintatticamente corrette del linguaggio. Si ottengono attraverso combinazione di formule atomiche, utilizzando i connettivi e i quantificatori. Sono definite ricorsivamente come segue:

- **Ogni atomo è una fbf**
- **Se A e B sono fbf, allora lo sono anche $\sim A$, $A \wedge B$, $A \vee B$, $A \leftrightarrow B$, $A \rightarrow B$** (eventualmente racchiuse tra parentesi tonde bilanciate)
- **Se A è una fbf e X è una variabile, $\exists X A$ e $\forall X A$ sono fbf**

Le espressioni (E1), (E2), (E3) sono formule ben formate, mentre non lo sono (E4) e (E5).

fbf in forma normale prenessa disgiuntiva (“disjunctive prenex normal form”): disgiunzione di una o più fbf composte da congiunzioni di letterali; le quantificazioni compaiono tutte in testa a F.

fbf in forma normale prenessa congiuntiva (“conjunctive prenex normal form”): congiunzione di una o più fbf composte da disgiunzioni di letterali; le quantificazioni compaiono tutte in testa ad F.

- Esempio
La fbf: $\exists x \forall y \exists z (a(x) \wedge b(y, z)) \vee (c(x) \wedge \sim a(z) \wedge d) \vee f$
è in forma normale disgiuntiva.
La fbf: $\exists x \forall y \exists z (a(x) \vee b(y, z)) \wedge (c(x) \vee \sim a(z) \vee d) \wedge f$
è in forma normale congiuntiva.

Qualunque fbf può essere trasformata in forma normale prenessa (congiuntiva o disgiuntiva) attraverso opportune trasformazioni sintattiche.

Letterale: fbf atomica o la sua negazione. Ad esempio, la formula (E1) è un letterale.

REGOLE DI PRECEDENZA TRA OPERATORI

1. $\sim \exists \forall$
 2. \wedge
 3. \vee
 4. $\rightarrow \leftrightarrow$
- Esempio
La fbf: $a \vee \sim b \wedge \exists x c(x) \rightarrow d(x, y)$
è equivalente a: $(a \vee (\sim b) \wedge (\exists x c(x))) \rightarrow d(x, y)$

Campo di azione (scope) di un quantificatore: fbf che lo segue immediatamente. Nel caso di ambiguità si utilizzano le parentesi tonde.

Esempio

- Nella fbf: $\forall x (p(x, y) \wedge q(x)) \vee q(x)$
- la quantificazione sulla variabile **x** ha come campo d'azione la formula $p(x, y) \wedge q(x)$

Variabili libere: variabili che non compaiono all'interno del campo di azione di un quantificatore. **Noi presupporremo sempre che le variabili sono quantificate o universalmente o esistenzialmente**, perché così è più semplice.

Formule chiuse: fbf che non contengono alcuna variabile libera. Ad esempio, le formule (E1), (E2) ed (E3) sono fbf chiuse.

Nel seguito considereremo solo formule fbf chiuse.

Formule ground: formule che non contengono variabili. Ad esempio la formula (E1) è una formula “ground”.

Varianti: una formula F2, ottenuta rinominando le variabili di una formula F1, è detta variante di F1.

7.2 - SEMANTICA

Occorre associare un significato ai simboli. Ogni sistema formale è la modellizzazione di una certa realtà (ad esempio la realtà matematica). O facciamo riferimento ad una semantica comune a tutti, oppure se ognuno usa la sua semantica, non ci si capisce. (Es. Attila → se per Attila intendo il tipo studiato in storia, ha due gambe. Se per Attila intendo il cane di William, ha 4 zampe. Semanticamente intendiamo un Attila diverso!)

Un'interpretazione è la costruzione di un rapporto fra i simboli del sistema formale e tale realtà (chiamata anche dominio del discorso).

Ogni formula atomica o composta della logica dei predicati del primo ordine può assumere il valore vero o falso in base alla frase che rappresenta nel dominio del discorso.

7.2.1 - INTERPRETAZIONE

Dato un linguaggio del primo ordine L un'interpretazione per L definisce un dominio non vuoto D e assegna:

- a ogni simbolo di costante in C, una costante in D
- a ogni simbolo di funzione n-ario F, una funzione $F: D^n \rightarrow D$
- a ogni simbolo di predicato n-ario in P una relazione in D^n , cioè un sottoinsieme di D^n

Es: Linguaggio del primo ordine, L, nel quale si ha una costante "0", un simbolo di funzione unaria "s" e un simbolo di predicato binario "p".

Es. Interpretazione I1 D: numeri naturali.

"0" rappresenta il numero zero, "s" rappresenta il successore di un numero naturale,

"p" rappresenta la relazione binaria " \leq "

Es. Interpretazione I2 D: numeri interi negativi.

"0" rappresenta il numero zero, "s" rappresenta il predecessore di un numero naturale,

"p" rappresenta la relazione binaria " \leq "

7.2.2 – VALORE DI VERITÀ

Data un'interpretazione il valore di verità di una fbf si definisce secondo le seguenti regole:

- **Formula atomica "ground" ha valore vero sotto un'interpretazione quando il corrispondente predicato è soddisfatto** (cioè quando la corrispondente relazione è vera nel dominio). La formula atomica ha valore falso quando il corrispondente predicato non è soddisfatto.
Es. Interpretazione I1 $\rightarrow p(0,s(0))$ vero poiché $0 \leq s(0) = 1$, $p(s(0), 0)$ falso
- **Formula composta:** il valore di verità di una formula composta rispetto a un'interpretazione si ottiene da quello delle sue componenti utilizzando le tavole di verità dei connettivi logici.
- **Formula quantificata esistenzialmente:** una formula del tipo $\exists X F$ è vera in un'interpretazione I se esiste almeno un elemento d del dominio D tale che la formula F' , ottenuta assegnando d alla variabile X, è vera in I. In caso contrario F ha valore falso.
- **Formula quantificata universalmente:** una formula del tipo $\forall X F$ è vera in un'interpretazione I se per ogni elemento d del dominio D, la formula F' , ottenuta da F sostituendo d alla variabile X, è vera in I. Altrimenti F ha valore falso.

Da notare che l'implicazione è sempre vera tranne quando l'antecedente è vero e il conseguente è falso.

A	B	$\neg A$	$A \wedge B$	$A \vee B$	$A \Rightarrow B$	$A \Leftrightarrow B$
T	T	F	T	T	T	T
T	F	F	F	T	F	F
F	T	T	F	T	T	F
F	F	T	F	F	T	T

Data la formula F: volano(asini) \Rightarrow ha_scritto(manzoni,promessi_sposi), assumendo l'interpretazione più intuitiva, F ha valore vero, poiché l'antecedente ha valore falso in tale interpretazione.

La formula F: $p(s(0),0) \Rightarrow p(0,s(0))$ ha valore vero nell'interpretazione I1 poiché l'antecedente ha valore falso, mentre ha valore falso in I2 poiché a un antecedente vero corrisponde un conseguente falso.

7.2.3 - MODELLI

Data una interpretazione I e una fbf chiusa F, I è un modello per F se e solo se F è vera in I.

(Es. Per la fbf $\forall Y p(0,Y)$ l'interpretazione I1 è un modello, mentre I2 non lo è.)

Una fbf è soddisfacibile se e solo se è vera almeno in una interpretazione, ovvero se esiste almeno un modello per essa.

Una fbf che ha valore vero per tutte le possibili interpretazioni, cioè per cui ogni possibile interpretazione è un modello, è detta logicamente valida.

Es. La fbf $\forall X p(X) \vee \neg (\forall Y p(Y))$ è logicamente valida. Infatti, le formule $\forall X p(X)$ e $\forall Y p(Y)$ sono semplici varianti della stessa formula F e quindi hanno i medesimi valori di verità per qualunque interpretazione. In generale, $F \vee \neg F$ ha sempre valore vero, in modo indipendente dall'interpretazione.

7.2.4 - INSIEMI DI FORMULE

Un insieme di formule chiuse del primo ordine S è soddisfacibile se esiste una interpretazione I che soddisfa tutte le formule di S (è un modello per ciascuna formula di S). Tale interpretazione è detta modello di S.

Es. $S = \{\forall Y p(Y,Y), p(s(0),0) \Rightarrow p(0,s(0))\}$.

L'interpretazione I1 è modello di S, mentre I2 non lo è. In I2 è infatti soddisfatta la prima formula dell'insieme, ma non la seconda.

Un insieme di formule S che non può essere soddisfatto da alcuna interpretazione, è detto insoddisfacibile (o inconsistente).

Es. l'insieme di formule $\{A, \neg A\}$ è insoddisfacibile.

7.2.5 - CONSEGUENZA LOGICA

Una formula F segue logicamente (o è conseguenza logica) da un insieme di formule S (e si scrive $S \models F$), se e solo se ogni interpretazione I che è un modello per S, è un modello per F.

Ci verrebbe da pensare che ci basta scrivere tutte le conseguenze logiche il cui fine è definire il sistema, e poi la nostra IA è finita. Il problema è che questo ha un costo computazionale veramente inaccettabile, quindi per forza dobbiamo trovare un altro modo.

Proprietà

- Se una fbf F segue logicamente da S ($S \models F$), allora l'insieme $S \cup \{\neg F\}$ è insoddisfacibile.
- Viceversa, se $S \cup \{\neg F\}$ è insoddisfacibile (e S era soddisfacibile), allora F segue logicamente da S.
- Difficile lavorare a livello semantico (interpretazione, modelli). Quindi si lavora a livello sintattico.

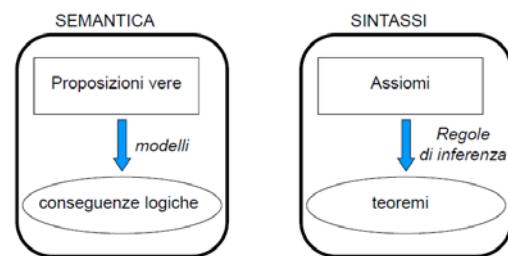
7.2.6 – SISTEMI DI REFUTAZIONE

I sistemi di refutazione si basano su questa proprietà: **per dimostrare $S \models F$ supposto S soddisfacibile è sufficiente dimostrare che $S \cup \{\neg F\}$ è insoddisfacibile.**

7.3 – TEORIE DEL PRIMO ORDINE

Esistono due apparati: quello semantico e quello sintattico.

Le teorie assiomatiche hanno dei criteri di trasformazione che producono da formule sintatticamente corrette, altre formule sintatticamente corrette. La dimostrazione è la sequenza di queste trasformazioni.



Quando si parla di calcolo si intende:

- **Calcolo proposizionale:** verifica di formula/e vera/e tramite le tavole di verità
- **Calcolo dei predicati del primo ordine:** tavole di verità troppo complesse. Dominio di interpretazione estremamente grande, se non infinito. Si ricorre al metodo assiomatico (noto come proof theory).

La logica dei predicati proposizionale e del primo ordine può essere formulata come sistema assiomatico-deduttivo.

Teoria assiomatica formata da:

- **Formule ben formate ritenute vere:** assiomi
- **Criteri di manipolazione sintattica:** regole di inferenza derivano fbf da fbf
- **Scopo:** produrre nuove formule sintatticamente corrette (**teoremi**).

Semplificazioni:

$$\begin{array}{lll}
 (A \wedge B) & \text{equivale a} & (\neg(A \rightarrow (\neg B))) \\
 (A \vee B) & \text{equivale a} & ((\neg A) \rightarrow B) \\
 (A = B) & \text{equivale a} & ((A \rightarrow B) \wedge (B \rightarrow A))
 \end{array}$$

Inoltre, per i quantificatori:

$$\begin{array}{lll}
 \exists X A & \text{abbrevia} & \neg(\forall X \neg A) \\
 \forall X A & \text{abbrevia} & \neg(\exists X \neg A)
 \end{array}$$

Proprietà fondamentali delle teorie del primo ordine:

- Dedicibilità
- Completezza
- Correttezza
- Monotonicità

7.3.1 – REGOLE DI INFERENZA

Modus Ponens (MP):

$$\frac{A, A \rightarrow B}{B}$$

Questa è una regola di inferenza molto nota. Le regole di inferenza hanno degli schemi che ci permettono di scrivere sintatticamente qualcosa, quindi questa mi dice che se ho una struttura A (che può essere qualunque formula ben formata) e ne ho un'altra che è $A \rightarrow B$, da questa posso derivare la formula B. (Es. Se Socrate è un uomo e l'uomo è mortale, allora Socrate è mortale.)

Specializzazione universale: se abbiamo una formula quantificata universalmente, es. $\forall x p(x)$, posso sempre derivare da questa una formula in cui al posto della x c'è un qualunque termine.

Specializzazione (Spec):

$$\frac{\forall X A}{A(t)}$$

Queste due regole di inferenza significano che se io voglio sapere se mortale(socrate) è un teorema, devo partire dalle mie formule e fare delle possibili trasformazioni che mi derivano il risultato sintattico mortale(socrate).

Una macchina deve trovare un albero di ricerca in cui gli operatori sono le regole di inferenza e alla fine in una foglia ci deve trovare il teorema mortale(socrate). Se lo trova è un teorema, se non lo trova non lo è.

Siamo tornati alla ricerca nello spazio degli stati ma qui gli operatori sono le regole di inferenza e non possiamo cancellare nulla se no perdiamo informazioni utili.

7.4 - DEDICIBILITÀ

Una teoria decidibile è una teoria per la quale esiste un metodo meccanico per stabilire se una qualunque fbf è un teorema o non lo è. Il calcolo dei predicati del primo ordine non è decidibile, ma semi-decidibile: se una formula è un teorema, esiste un metodo meccanico che la deriva in un numero finito di passi. Se invece la formula non è un teorema, non è garantita, in generale, la terminazione del metodo meccanico (Turing 1936, Church 1936).

Una teoria del primo ordine è un insieme di fbf chiuse (assiomi) e si può quindi parlare di modello di una teoria. Un modello per una teoria del primo ordine T è un'interpretazione che soddisfa tutti gli assiomi di T (assiomi logici e assiomi propri). Se T ha almeno un modello viene detta consistente (o soddisfacibile).

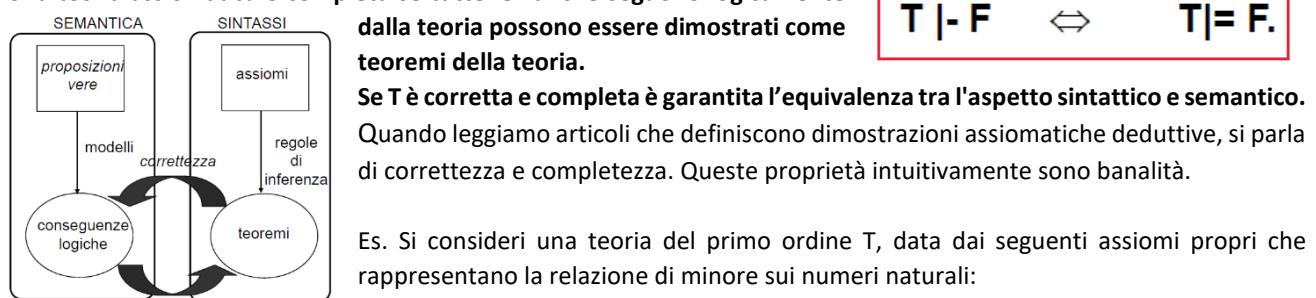
7.5 – CORRETTEZZA E COMPLETEZZA

Una teoria assiomatica è corretta se i teoremi dimostrati seguono logicamente dagli assiomi della teoria.

Una teoria assiomatica è completa se tutte le fbf che seguono logicamente

dalla teoria possono essere dimostrati come teoremi della teoria.

$$T \vdash F \Leftrightarrow T \models F.$$



Se T è corretta e completa è garantita l'equivalenza tra l'aspetto sintattico e semantico.

Quando leggiamo articoli che definiscono dimostrazioni assiomatiche deduttive, si parla di correttezza e completezza. Queste proprietà intuitivamente sono banalità.

Es. Si consideri una teoria del primo ordine T, data dai seguenti assiomi propri che rappresentano la relazione di minore sui numeri naturali:

$$\begin{aligned} & \forall X \forall Y (p(X, Y) \rightarrow p(X, s(Y))) & (A1) \\ & \forall X p(X, s(X)) & (A2) \\ & & (A3) \end{aligned}$$

Le regole di inferenza di T siano Modus Ponens, Specializzazione e la seguente regola:

In T si deriva come teorema la formula $p(0,0)$ applicando le seguenti trasformazioni:

Da Spec. e A2: $\forall X \forall Y (p(X, Y) \rightarrow p(X, s(Y))) \Rightarrow \forall Y (p(0, Y) \rightarrow p(0, s(Y)))$ (T1)

Abduzione (ABD):

$$\frac{B, A \rightarrow B}{A}$$

Da Spec. e T1: $p(0,0) \rightarrow p(0,s(0))$

(T2)

Applicando ABD a T2 e A6: $p(0,0)$

(T5)

A causa dell'applicazione dell'abduzione, questa teoria non è corretta: un'interpretazione che ha come dominio l'insieme dei numeri naturali e associa al simbolo di funzione "s" la funzione successore e al simbolo di predicato "p" la relazione < (minore) è un modello per gli assiomi, ma non per la formula $p(0,0)$.

Es. sta-male(mario). $\forall X (ha-epatite(X) \rightarrow sta-male(X))$.

Si conclude: $ha-epatite(mario)$. → **ERRORE!!**

7.6 – MONOTONICITÀ

Un'altra proprietà fondamentale delle teorie del primo ordine è la monotonicità. **Una teoria T è monotona se l'aggiunta di nuovi assiomi non invalida i teoremi trovati precedentemente.**

7.6.1 - PROPRIETÀ

- Sia $Th(T)$ l'insieme dei teoremi derivabili da T. **Allora T è monotona se $Th(T) \subseteq Th(T \cup H)$ per qualunque insieme aggiuntivo di assiomi H.**
- Esistono regole di inferenza non monotone, ad esempio la regola nota come **Assunzione di Mondo Chiuso ("Closed World Assumption" o CWA):**

$$\frac{T \models \neg A}{\sim A}$$

Se una formula atomica "ground" A non è conseguenza logica di una teoria T, $\sim A$ si può considerare un teorema di T. Se alla teoria T si aggiunge l'assimmo A, non si può più derivare $\sim A$, da cui segue la non monotonicità del sistema di inferenza.

8 - INFERENZA E LOGICA PROPOZIONALE

La logica proposizionale è la logica più semplice, ma non molto espressiva. Non possiamo esprimere variabili (solo enumerazione di tutti gli elementi). Infatti, se S, S1 e S2 sono sentenze, allora sono anche sentenze:

$$\begin{array}{lll} \neg S \text{ (negazione)} & S_1 \wedge S_2 \text{ (congiunzione)} & S_1 \Leftrightarrow S_2 \text{ (bicondizionale)} \\ S_1 \vee S_2 \text{ (disgiunzione)} & S_1 \Rightarrow S_2 \text{ (implicazione)} & \end{array}$$

Una qualunque fbf della logica proposizionale si può trasformare in un equivalente insieme di clausole generali.

8.1 - CLAUSOLE

Una clausola è una disgiunzione di letterali (cioè formule atomiche negate e non negate), **in cui tutte le variabili sono quantificate universalmente in modo implicito**.

Una clausola generica può essere rappresentata come la **disgiunzione**: $A_1 \vee A_2 \vee \dots \vee A_n \vee \neg B_1 \vee \dots \vee \neg B_m$ dove A_i ($i=1, \dots, n$) e B_j ($j=1, \dots, m$) sono atomi.

Una clausola nella quale non compare alcun letterale, sia positivo sia negativo, è detta **clausola vuota** e verrà indicata con \square . Interpretato come contraddizione: disgiunzione falso $\vee \neg$ vero

Un sottoinsieme delle clausole è costituito dalle **clausole definite**, nelle quali si ha sempre un solo letterale positivo: $A_1 \vee \neg B_1 \vee \dots \vee \neg B_m$

8.2 - PRINCIPIO DI RISOLUZIONE

Il principio di risoluzione si applica a teorie del primo ordine in forma a clausole ed è la **regola di inferenza base utilizzata nella programmazione logica**.

Il principio di risoluzione, che si applica a formule in forma a clausole, è utilizzato dalla maggior parte dei risolutori automatici di teoremi. Nella logica proposizionale abbiamo clausole prive di variabili.

Siano C_1 e C_2 due clausole prive di variabili: $C_1 = A_1 \vee \dots \vee A_n$, $C_2 = B_1 \vee \dots \vee B_m$

Se esistono in **C1 e C2 due letterali opposti**, A_i e B_j tali che $A_i = \neg B_j$, allora da C_1 e C_2 (**clausole parent**) si può derivare una nuova clausola C_3 , denominata **risolvente** (che è una regola di produzione), della forma:

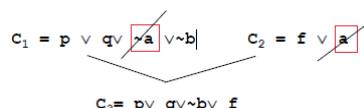
$C_3 = A_1 \vee \dots \vee A_{i-1} \vee A_{i+1} \vee \dots \vee A_n \vee B_1 \vee \dots \vee B_{j-1} \vee B_{j+1} \vee \dots \vee B_m \rightarrow C_3$ è conseguenza logica di $C_1 \cup C_2$.

Es. Suppongo di avere una base di conoscenza composta da C_1 e C_2 , l'importante è che siano sintatticamente corrette. Una è un atomo, l'altra è una disgiunzione, quindi sono entrambe clausole.

Da queste due viene derivata C_3 e siccome $p(0,0)$ c'è in tutte due in forma complementare, elido questo contributo e mi rimane solo $p(0, s(0))$.



Es. Suppongo di avere C_1 e C_2 . Siccome $A \rightarrow B$, allora $B \vee \neg A$ (B or not A). Da A e $A \rightarrow B$, derivo B .



8.2.1 - DIMOSTRAZIONE

Dimostrazione per contraddizione attraverso la risoluzione: dati gli assiomi propri H di una teoria e una formula F , derivando da $H \cup \neg F$ la contraddizione logica si dimostra che F è un teorema della teoria.

1. Ridurre H e il teorema negato $\neg F$ in forma a clausole
 - a. H trasformato nell'insieme di clausole H^c : $H \rightarrow H^c$
 - b. F negata e trasformata nell'insieme di clausole F^c : $\neg F \rightarrow F^c$

All'insieme $H^c \cup F^c$ si applica la risoluzione: se F è un teorema della teoria, allora la risoluzione deriva la contraddizione logica (clausola vuota) in un numero finito di passi.

Contraddizione: Nella derivazione compariranno due clausole del tipo A e $\neg A$ con A e B formule atomiche unificabili.

Se ho A e $\neg A$ e applico la risoluzione, è ovvio che non trovo nulla perché il sistema è contraddittorio, trovo la clausola vuota. In questo caso quello che vuole dimostrare, in realtà lo nega perché immagina che sia falso, poi lo aggiunge alla sua base: è un ragionamento per assurdo.

Poi cercherà la clausola vuota: se trova la contraddizione logica, quella cosa lì non può essere falsa, quindi è vera, altrimenti se non trova la contraddizione logica, quello non è un teorema.

Se è così, però, dobbiamo dare una condizione sulla base di conoscenza, questa deve essere NON contraddittoria poiché se fosse già contraddittoria di partenza, il mio sistema di risoluzione inserendo la negazione del problema, arriverebbe alla contraddizione logica comunque, ma ci arriverebbe anche senza aggiungere la nostra negazione perché di per sé è già contraddittoria, quindi ci direbbe sempre "sì, ho trovato la contraddizione".

Noi quando diciamo che una persona si contraddice? "oggi piove?" risposta "sì", "oggi non piove?" risposta "sì": il nostro sistema direbbe sì sia ad A sia a $\neg A$.

Es.

$H = \{(a \rightarrow c \vee d) \wedge (a \vee d \vee e) \wedge (a \rightarrow \neg c)\}$
 $F = \{d \vee e\}$ (formula da dimostrare)
• La trasformazione in clausole di H e $\neg F$ produce:
 $H^c = \{\neg a \vee c \vee d, a \vee d \vee e, \neg a \rightarrow \neg c\}$
 $F^c = \{\neg d, \neg e\}$ (cioè $\neg(d \vee e)$)
• Si vuole dimostrare che $H^c \cup F^c$:
 $\begin{array}{ll} \{\neg a \vee c \vee d, & (1) \\ a \vee d \vee e, & (2) \\ \neg a \vee \neg c, & (3) \\ \neg d, & (4) \\ \neg e\} & (5) \end{array}$
• è contraddittorio.
Ho applicato De Morgan

$A \rightarrow c \vee d = \neg a \vee c \vee d$
 $A \rightarrow \neg c = \neg a \vee \neg d$

$\neg(d \vee e)$ non è una formula normale, devo portare il negato a ridosso degli atomi. Bisogna applicare de Morgan e arriviamo a $(\neg d \text{ and } \neg e)$

Aggiungo not d e not e

Tutti i possibili risolventi al passo 1 sono:

$\{c \vee d \vee e\}$	(6)	da (1) e (2)
$\{d \vee e \vee \neg c\}$	(7)	da (2) e (3)
$\{\neg a \vee c\}$	(8)	da (1) e (4)
$\{a \vee e\}$	(9)	da (2) e (4)
$\{a \vee d\}$	(10)	da (2) e (5)
$\{\neg a \vee d\}$	(11)	da (1) e (3)

Al passo 2, da (10) e (11) viene derivato il risolvente: d (12)

Al passo 3, da (4) e (12) viene derivata anche la clausola vuota.

8.3 - ALGORITMO DI RISOLUZIONE PER LOGICA PROPOZIZIONALE

```

function PL-RESOLUTION( $KB, \alpha$ ) returns true or false
  clauses  $\leftarrow$  the set of clauses in the CNF representation of  $KB \wedge \neg \alpha$ 
  new  $\leftarrow \{\}$ 
  loop do
    for each  $C_i, C_j$  in clauses do
      resolvents  $\leftarrow$  PL-RESOLVE( $C_i, C_j$ )
      if resolvents contains the empty clause then return true
      new  $\leftarrow$  new  $\cup$  resolvents
    if new  $\subseteq$  clauses then return false
    clauses  $\leftarrow$  clauses  $\cup$  new
  end loop
end function

```

8.4 - FORWARD CHAINING E BACKWARD CHAINING

Sono algoritmi molto naturali e con complessità lineare in tempo. Essi sono **completi per le clausole di Horn**. **Forward Chaining** è **data-driven** (es. monitoring, configurazione, interpretazione) e non si concentra sull'obiettivo, mentre **Backward Chaining** è **goal-driven**, quindi in generale è più appropriato per il problem-solving.

9 - INFERNZA E LOGICA DEI PREDICATI

La logica dei predicati è un linguaggio più potente per esprimere KB (variabili e quantificatori).

9.1 - TRASFORMAZIONE IN CLAUSOLE

Le clausole generali nella logica del primo ordine sono le stesse presentate prima, però questa volta questi letterali possono contenere delle variabili. In questo caso le clausole generali conterranno variabili e si presuppone che queste siano quantificate universalmente, quindi nelle clausole generali le variabili ci sono ma sono tutte quantificate universalmente, quindi il quantificatore non c'è (perché è universale).

Qua il principio di risoluzione lavora solo con formule in forma di clausole in cui i quantificatori sono tutti universali, quindi è possibile trasformare una qualunque formula della logica del primo ordine con variabili quantificate universalmente nella formula generale. I passi sintattici che vedremo sono quelli che compie la macchina in modo automatico.

Una qualunque fbf della logica dei predicati si può trasformare in un insieme di clausole generali. Passi:

1. Trasformazione in fbf chiusa
2. Applicazione delle equivalenze per i connettivi logici
3. Applicazione della negazione ad atomi e non a formule composte
4. Cambiamento di nomi delle variabili
5. Spostamento dei quantificatori in testa alla formula (forma prenessa)
6. Forma normale congiuntiva
7. Skolemizzazione
8. Eliminazione dei quantificatori universali

Qualunque teoria del primo ordine T può essere trasformata in una teoria T' in forma a clausole.

Anche se T non è logicamente equivalente a T' a causa dell'introduzione delle funzioni di Skolem, vale comunque la seguente **proprietà**: sia T una teoria del primo ordine, T' una sua trasformazione in clausole. Allora T è insoddisfacibile se e solo se T' è insoddisfacibile.

Il principio di risoluzione è una procedura di dimostrazione che opera per contraddizione e si basa sul concetto di insoddisfabilità.

9.1.1 – TRASFORMAZIONE IN FBF CHIUSA

Es. Una formula aperta a noi non interessa, quindi dobbiamo renderla chiusa perché deve essere o esistenziale o universale.

$$\forall X (p(Y) \rightarrow \neg(\forall Y (q(X,Y) \rightarrow p(Y)))) \quad (1)$$

è trasformata in:

$$\forall X \exists Y (p(Y) \rightarrow \neg(\forall Y (q(X,Y) \rightarrow p(Y)))) \quad (2)$$

La (2) è un'implicazione dentro un'implicazione con dei quantificatori nelle variabili. In linguaggio naturale: se per qualunque x vale questa proprietà allora vuol dire che diventa vero che per qualunque altro y vale anche quest'altra proprietà.

9.1.2 – APPLICAZIONE EQUIVALENZE PER CONNETTIVI LOGICI

Es. $A \rightarrow B$ è sostituito da $\neg A \vee B$

$$\forall X \forall Y (\neg p(Y) \vee \neg(\forall Y (\neg q(X,Y) \vee p(Y)))) \quad (3)$$

9.1.3 – APPLICAZIONE NEGAZIONE AD ATOMI E NON A FORMULE COMPOSTE

Sappiamo con le Leggi di De Morgan che:

$$\begin{aligned} \forall X \neg A &\text{ equivale a } \neg \exists X A \\ \exists X \neg A &\text{ equivale a } \neg \forall X A \\ \neg(A_1 \vee A_2 \vee \dots \vee A_n) &\text{ equivale a } \neg A_1 \wedge \neg A_2 \wedge \dots \wedge \neg A_n \\ \neg(A_1 \wedge A_2 \wedge \dots \wedge A_n) &\text{ equivale a } \neg A_1 \vee \neg A_2 \vee \dots \vee \neg A_n \end{aligned}$$

Allora

$$\forall X \forall Y (\neg p(Y) \vee (\exists Y (q(X,Y) \wedge \neg p(Y)))) \quad (4)$$

9.1.4 – CAMBIAMENTO NOMI VARIABILI

In (4) abbiamo un esiste Y, ma visto che la formula delle clausole generali ha tutte le variabili quantificate universalmente, io devo portare tutti i quantificatori in testa. Se io porto in testa l'esiste Y, non si capisce più di cosa stiamo parlando perché c'è un altro Y. Allora visto che lo scope della variabile è delimitato, quando porto tutto in testa faccio un rename, se no c'è ambiguità.

$$\forall X \forall Y (\neg p(Y) \vee (\exists Z (q(X,Z) \wedge \neg p(Z)))) \quad (5)$$

9.1.5 – SPOSTAMENTO DEI QUANTIFICATORI

In questa fase si spostano tutti i quantificatori in testa alla formula, rendendola una formula in forma prenessa.

$$\forall X \forall Y \exists Z (\neg p(Y) \vee ((q(X,Z) \wedge \neg p(Z)))) \quad (6)$$

9.1.6 – FORMA NORMALE CONGIUNTIVA

Qui si applica la proprietà distributiva e si rende la formula in forma congiuntiva, cioè come congiunzione di disgiunzioni, sempre con quantificazione in testa.

$$\forall X \forall Y \exists Z (\neg p(Y) \vee ((q(X,Z) \wedge (\neg p(Y) \vee \neg p(Z)))) \quad (7)$$

9.1.7 - SKOLEMIZZAZIONE

Ogni variabile quantificata esistenzialmente viene sostituita da una funzione delle variabili quantificate universalmente che la precedono. Tale **funzione** è detta **di Skolem**.

Es. del linguaggio naturale: io dico "oggi sono andata a pagare la luce, poi arriva tizio e mi si piazza davanti".

Significato: esiste una persona che è arrivata e si è piazzata prima di me, quindi tizio è una costante, è un nome, però nella mia testa la costante è equivalente ad un esistenziale.

Tizio è un nome fittizio, in generale noi usiamo nel linguaggio naturale delle costanti per in realtà esprimere degli esistenziali, e infatti sono nomi strani.

L'idea deriva da Skolem, che è un logico, che al posto delle variabili esistenziali dice che possiamo mettere delle costanti che devono essere diverse da ogni costante che ha un significato nel nostro dominio, non facciamo proprio la stessa cosa ma è simile.

Se dico "mario mi si è piazzato davanti", mario lo conosco, è una costante che ha un significato, è diverso.

Es. Io dovrei dire che per qualunque x esiste una madre di x, quindi al posto della costante devo metterci un funtore che però dipende da x, perché se uso la costante, significa che tutti hanno la stessa madre, ma questo non è vero. Perciò "esiste Y per ogni X madre(Y,X)" cioè per qualunque essere esiste Dio e immagino che questo Dio sia uno, allora se c'è un solo Dio per tutti, dirò che esiste un Y che vale per qualunque X, quindi quella diventa una costante uguale per tutti.

In questo caso generalizzando, si tolgono gli esistenziali mettendo al loro posto un funtore con delle variabili che sono quelle che sono nello scope di quell'esistenziale; **non c'è il problema della valutazione perché ora stiamo ragionando solo sintatticamente.**

In (7) Z è sostituita da f(X,Y), perché Z si trova nel campo di azione delle quantificazioni $\forall X$ e $\forall Y$:

$$\forall X \forall Y \exists Z (\neg p(Y) \vee ((q(X,f(X,Y)) \wedge (\neg p(Y) \vee \neg p(f(X,Y))))) \quad (8)$$

9.1.8 – ELIMINAZIONE DEI QUANTIFICATORI UNIVERSALI

Si ottiene una formula detta universale perché tutte le sue variabili sono quantificate universalmente ed essa è in forma normale congiuntiva.

$$\forall X \forall Y (\neg p(Y) \vee (q(X,f(X,Y)) \wedge (\neg p(Y) \vee \neg p(f(X,Y))))) \quad (9)$$

Una formula di questo tipo rappresenta un insieme di clausole, ciascuna da un congiunto nella formula. La forma normale a clausole che si ottiene è:

$$\{ \sim p(Y) \vee q(X, f(X, Y)), \sim p(Y) \vee \sim p(f(X, Y)) \} \quad (10)$$

La seconda clausola può essere riscritta rinominando le variabili, sostituendo la formula con una sua variante:

$$\{ \sim p(Y) \vee q(X, f(X, Y)), \sim p(Z) \vee \sim p(f(W, Z)) \} \quad (11)$$

9.2 - UNIFICAZIONE

L'unificazione è un **procedimento di manipolazione formale utilizzato per stabilire quando due espressioni possono coincidere procedendo a opportune sostituzioni.**

Sostituzione: σ insieme di legami di termini T_i a simboli di variabili X_i ($i=1, \dots, n$).

$\sigma = \{X_1/T_1, X_2/T_2, \dots, X_n/T_n\}$ dove X_1, X_2, \dots, X_n sono distinte ($x/T = x$ legato al termine T)

La sostituzione corrispondente all'insieme vuoto è detta sostituzione identità ϵ .

9.3 – SOSTITUZIONI E RENAMING

Applicare la sostituzione σ a un'espressione E , $[E]\sigma$ o istanza di E , produce una nuova espressione ottenuta sostituendo simultaneamente ciascuna variabile X_i dell'espressione con il corrispondente termine T_i .

Renaming: sostituzioni che cambiano semplicemente il nome ad alcune delle variabili di E , $[E]\sigma$ in questo caso è una variante di E .

Le sostituzioni si possono anche combinare, infatti se ho σ_1 e σ_2 :

$$\sigma_1=\{X_1/T_1, X_2/T_2, \dots, X_n/T_n\} \quad \sigma_2=\{Y_1/Q_1, Y_2/Q_2, \dots, Y_m/Q_m\}$$

La combinazione di sostituzioni è $\sigma_1\sigma_2$:

$$\{X_1/[T_1]\sigma_2, \dots, X_n/[T_n]\sigma_2, Y_1/Q_1, Y_2/Q_2, \dots, Y_m/Q_m\}$$

Cancellando le coppie $X_i/[T_i]\sigma_2$ per le quali si ha $X_i=[T_i]\sigma_2$ e le coppie Y_j/Q_j per le quali Y_j appartiene all'insieme $\{X_1, X_2, \dots, X_n\}$.

Più in generale, una sostituzione θ è più generale di una sostituzione σ se esiste una sostituzione λ che $\sigma=\theta\lambda$.

Es: La sostituzione $\theta=\{Y/T, Z/neri\}$ è più generale della sostituzione $\sigma=\{Y/bianchi, T/bianchi, Z/neri\}$ in quanto si può ottenere attraverso la composizione $\{Y/T, Z/neri\} \{T/bianchi\}$ con $\sigma=\theta\lambda$, e $\lambda=\{T/bianchi\}$.

9.4 - SOSTITUZIONE UNIFICATRICE

L'unificazione rende identici due o più atomi/termini (o meglio le loro istanze) attraverso un'opportuna sostituzione.

Se si considerano solo due atomi/termini, uno dei quali senza alcuna variabile, si ricade in un caso particolare di unificazione, detto **pattern-matching**.

Un insieme di atomi/termini A_1, A_2, \dots, A_n è unificabile se esiste una sostituzione σ tale che: $[A_1]\sigma=[A_2]\sigma=\dots=[A_n]\sigma$ e la sostituzione σ è detta **sostituzione unificatrice (o unificatore)**.

Es. Se si considerano gli atomi: $A_1=c(Y, Z)$ $A_2=c(T, neri)$

Possibili sostituzioni unificatrici sono: $\sigma=\{Y/T, Z/neri\}$ $\theta=\{Y/bianchi, T/bianchi, Z/neri\}$

La loro applicazione produce la stessa istanza: $[c(Y, Z)]\theta=[c(T, neri)]\theta=c(T, neri)$

$$[c(Y, Z)]\sigma=[c(T, neri)]\sigma=c(bianchi, neri)$$

La sostituzione: $\lambda=\{Y/T, Z/bianchi\}$ non è un unificatore per A_1 e A_2 perché produce istanze diverse.

Es. Per gli atomi: $A_3=p(X, X, f(a, Z))$ $A_4: p(Y, W, f(Y, J))$

Possibili sostituzioni unificatrici sono: $\sigma=\{X/a, Y/a, W/a, J/Z\}$ $\theta=\{X/a, Y/a, W/a, J/c, Z/c\}$

La loro applicazione ad A_3 e A_4 produce la stessa istanza:

$p(a, a, f(a, Z))$ nel caso della sostituzione σ

$p(a, a, f(a, c))$ nel caso della sostituzione θ

Possono esistere più sostituzioni unificatrici, si vuole individuare quella più generale (**MGU most general unifier**).

σ mgu: θ si ottiene da σ componendola con $\lambda=\{Z/c\}$. (**Attenzione perché MGU non è detto che esista sempre**)

9.5 - ALGORITMO DI UNIFICAZIONE

Algoritmo in grado di determinare se due atomi/termini sono unificabili o meno e restituire, nel primo caso, la sostituzione unificatrice più generale. Esistono vari algoritmi di unificazione di differente complessità. **Regole alla base dell'algoritmo di unificazione fra due termini T1 e T2:**

T1 \ T2	<costante> C1	<costante> C2	<variabile> X1	<variabile> X2	<termine composto> S1	<termine composto> S2
<costante> C1	ok se C1=C2		ok $\{X2/C1\}$		NO	
<variabile> X1		ok $\{X1/C2\}$		ok $\{X1/X2\}$		ok $\{X1/S2\}$
<termine composto> S1			NO	ok $\{X2/S1\}$		ok se S1 e S2 hanno stesso funtore e arietà e gli argomenti UNIFICANO

La funzione termina sempre ed è in grado di fornire o la sostituzione più generale per unificare A e B o un fallimento (FALSE). Un termine composto (cioè diverso da costante o variabile) è rappresentato da un operatore OP (il simbolo funzionale del termine) e come altri elementi gli argomenti del termine ARGS (lista). La funzione head, applicata a una lista L, restituisce il primo elemento di L, mentre la funzione tail il resto della lista L.

```
function UNIFY(x, y, θ) returns a substitution to make x and y identical
  inputs: x, a variable, constant, list, or compound
          y, a variable, constant, list, or compound
          θ, the substitution built up so far

  if θ = failure then return failure
  else if x = y then return θ
  else if VARIABLE?(x) then return UNIFY-VAR(x, y, θ)
  else if VARIABLE?(y) then return UNIFY-VAR(y, x, θ)
  else if COMPOUND?(x) and COMPOUND?(y) then
    return UNIFY(ARGS[x], ARGS[y], UNIFY(OP[x], OP[y], θ))
  else if LIST?(x) and LIST?(y) then
    return UNIFY(REST[x], REST[y], UNIFY(FIRST[x], FIRST[y], θ))
  else return failure
```

```
function UNIFY-VAR(var, x, θ) returns a substitution
  inputs: var, a variable
          x, any expression
          θ, the substitution built up so far

  if {var/val} ∈ θ then return UNIFY(val, x, θ)
  else if {x/val} ∈ θ then return UNIFY(var, val, θ)
  else if OCCUR-CHECK?(var, x) then return failure
  else return add {var/x} to θ
```

9.6 - OCCUR CHECK

È il controllo che un termine variabile da unificare con un secondo termine non compaia in quest'ultimo. Necessario per assicurare la terminazione dell'algoritmo e la correttezza del procedimento di unificazione.

I due termini "X" e "f(X)" non sono corretti perché non esiste una sostituzione per X che renda uguali i due termini. Se un termine t ha una struttura complessa, la verifica se X compare in t può essere anche molto inefficiente.

Prolog non utilizza l'occur-check: quindi non è corretto. È molto costoso, per questo non è implementato perché dopo un po' c'è un loop perché l'ultima variabile va a richiamare il punto di partenza.

9.7 - PRINCIPIO DI RISOLUZIONE PER CLAUSOLE GENERALI

Siano C1 e C2 due clausole del tipo: C1=A1 $\vee \dots \vee$ An, C2=B1 $\vee \dots \vee$ Bm (la forma è sempre quella, cambia che adesso ci possono essere delle variabili) dove Ai (i=1..n) e Bj (j=1..m) sono letterali positivi o negativi in cui possono comparire variabili.

Se esiste Ai e Bj tali che [Ai] $\theta = [\neg Bj]\theta$, dove θ è la sostituzione unificatrice più generale, allora si può derivare una nuova clausola C3 (il risolvente): [A1 $\vee \dots \vee$ Ai-1 \vee Ai+1 $\vee \dots \vee$ An \vee B1 $\vee \dots \vee$ Bj-1 \vee Bj+1 $\vee \dots \vee$ Bm] θ .

Date due clausole C1 e C2, il loro risolvente C3 è conseguenza logica di C1UC2.

Es. p(a) or p(b), not p(x) or g(x, f)

Applico il principio di risoluzione e trovo due atomi complementari. La sostituzione unificatrice è $\alpha\{x/a\}$.

La nuova clausola è p(b) or g(x, f), ma al posto di x ci metto a.

Risultato: p(b) or g(a, f)

9.7.1 – MGU PER RICAVARE IL RISOLVENTE

$$\frac{A_1 \vee \dots \vee A_n B_1 \vee \dots \vee B_m \exists \theta : [A_i]\theta = [\neg B_j]\theta}{[A_1 \vee \dots \vee A_{i-1} \vee A_i +_1 \dots \vee A_n \vee B_1 \vee \dots \vee B_{j-1} \vee B_{j+1} \vee \dots \vee B_m]\theta}$$

Dove θ è la sostituzione più generale per Ai e $\neg Bj$.

Es. Insieme H= { $\forall X$ (uomo(X) \rightarrow mortale(X)), uomo(socrate)}.

Insieme F = { $\exists X$ mortale(X)}.

(Esiste un x mortale? Prima nego la formula e scrivo: not esiste x mortale(x). Ora non è in forma clausole, allora sposto il not a ridosso del predicato cambiando il quantificatore: qualunque x not mortale x. Poi qualunque lo butto e mi ritrovo: not mortale(X))

La trasformazione in clausole di H e $\neg F$ produce:

HC = { \neg uomo(X) \vee mortale(X), uomo(socrate)}

FC = { \neg mortale(X)}

L'insieme HC U FC è il seguente:

{ \neg uomo(X) \vee mortale(X) (1)

uomo(socrate) (2)

\neg mortale(Y) (3)

Nella (2) x deve essere legato a socrate, quindi ho mortale(socrate) anche in aggiunta. Ora posso prendere mortale(socrate) e not mortale(y): lego y a socrate e trovo il risolvente. Chi è il risolvente? Uno è mortale(socrate), l'altro è not mortale(y), quindi la nuova clausola è una clausola vuota: ho trovato la contraddizione. Esiste allora un x mortale? Il sistema risponde sì. Il sistema si tiene in pancia un'informazione importante che posso estrarre ed è "questo è successo legando x a socrate".

La variabile X di FC rinominata con Y che non appare in nessuna altra clausola dell'insieme. Questa operazione viene eseguita ogni volta che si aggiungono nuovi risolventi all'insieme delle clausole. Al passo 1, tutti i possibili risolventi, e le sostituzioni unificatrici più generali applicate per derivarli, sono:

{mortale(socrate) (4) $\theta = \{X/\text{socrate}\}$

\neg uomo(Z) (5) $\theta = \{X/Y\}$

Al passo 2, da (2) e (5) viene derivata la clausola vuota, applicando la sostituzione {Z/socrate}. La clausola vuota è anche derivabile dalle clausole (3) e (4) applicando la sostituzione {Y/socrate}.

Es.

1. cane(fido). 2. \neg abbaia(fido).

3. scodinzola(fido). 4. miagola(geo).

5. scodinzola(X) and cane(X) \rightarrow amichevole(X). Tradotto: \neg scodinzola(X) or \neg cane(X) or amichevole(X).

6. amichevole(X1) and \neg abbaia(X1) \rightarrow \neg spaventato(Y1,X1).

Tradotto: \neg amichevole(X1) or abbaia(X1) or \neg spaventato(Y1,X1).

7. cane(X2) \rightarrow animale(X2). Tradotto: $\sim\text{cane}(X2)$ or $\text{animale}(X2)$.
 8. miagola(X3) \rightarrow gatto(X3). Tradotto: $\sim\text{miagola}(X3)$ or $\text{gatto}(X3)$.
 9. esiste X4 Y cane(X4) and gatto(Y) and $\sim\text{spaventato}(Y,X4)$. **GOAL**
 Tradotto: $\sim\text{cane}(X4)$ or $\sim\text{gatto}(Y)$ or $\text{spaventato}(Y,X4)$.

- Da 9 e 1: 10. $\sim\text{gatto}(Y)$ or $\text{spaventato}(Y,\text{fido})$.
 Da 10. e 6: 11. $\sim\text{amichevole}(\text{fido})$ or $\text{abbaia}(\text{fido})$ or $\sim\text{gatto}(Y)$.
 Da 11. e 2: 12. $\sim\text{amichevole}(\text{fido})$ or $\sim\text{gatto}(Y)$.
 Da 8. e 12: 13. $\sim\text{amichevole}(\text{fido})$ or $\sim\text{miagola}(Y)$.
 Da 13. e 4: 14. $\sim\text{amichevole}(\text{fido})$.
 Da 5. e 14: 15. $\sim\text{scodinzola}(\text{fido})$ or $\sim\text{cane}(\text{fido})$.
 Da 3. e 15: 16. $\sim\text{cane}(\text{fido})$
 Da 1. e 16. CONTRADDIZIONE!

9.8 - CORRETTEZZA E COMPLETEZZA

Si può dimostrare che **sotto opportune strategie, la risoluzione è corretta e completa**.

Se viene generata la clausola vuota, la teoria HC U FC è insoddisfacibile e se la teoria HC U FC è insoddisfacibile, la derivazione genera la clausola vuota in un numero finito di passi.

Teorema (Correttezza e completezza della risoluzione)

Un insieme di clausole è insoddisfacibile se e solo se l'algoritmo di risoluzione termina con successo in un numero finito di passi, generando la clausola vuota.

Il metodo di risoluzione procede esaustivamente generando tutti i possibili risolventi ad ogni passo.

9.9 - STRATEGIE

Si definiscono **strategie** che scelgono opportunamente le clausole da cui derivare un risolvente.

I metodi di prova che si ottengono risultano **più efficienti anche se** in alcuni casi **possono introdurre incompletezza**. La dimostrazione attraverso il principio di risoluzione può essere rappresentata con un grafo, detto **grafo di refutazione**. Le **clausole** dell'insieme base HC U FC sono **nodi del grafo** dai quali **possono solo uscire archi**. Un **risolvente** corrisponde a un **nodo nel quale entrano almeno due archi** (ciascuno da una delle due clausole "parent").

Strategia in ampiezza ("breadth-first"). Al passo i ($i \geq 0$), genera tutti i possibili risolventi a livello $i+1$ -esimo utilizzando come clausole "parent" una clausola di C_i (cioè una clausola a livello i) e una di C_j ($j \leq i$), cioè una clausola appartenente a un livello uguale o minore di i .

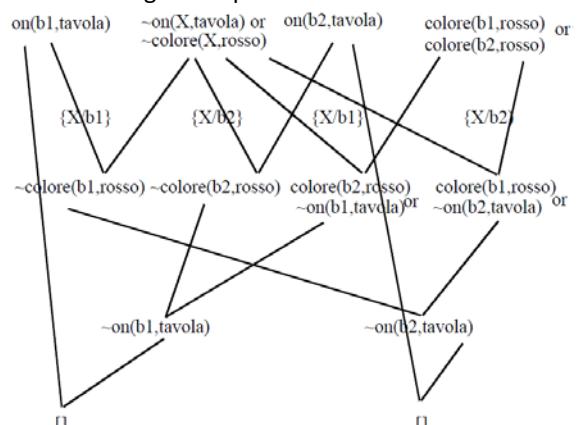
Es.

Quello che vogliamo dimostrare è che la quarta clausola deve essere PRIMA negata mettendo un not all'esterno di "esiste x ", poi quella non è una clausola generale perché non ha la negazione a ridosso degli atomi e ha un and (mentre le clausole possono avere solo or).

A questo punto c'è un not esiste x , quindi sposto l'esistenziale facendo diventare un universale, quindi diventa: per qualunque x , not esiste x . Ora applico de Morgan e lo aggiungo alla mia knowledge base per cercare la clausola vuota .

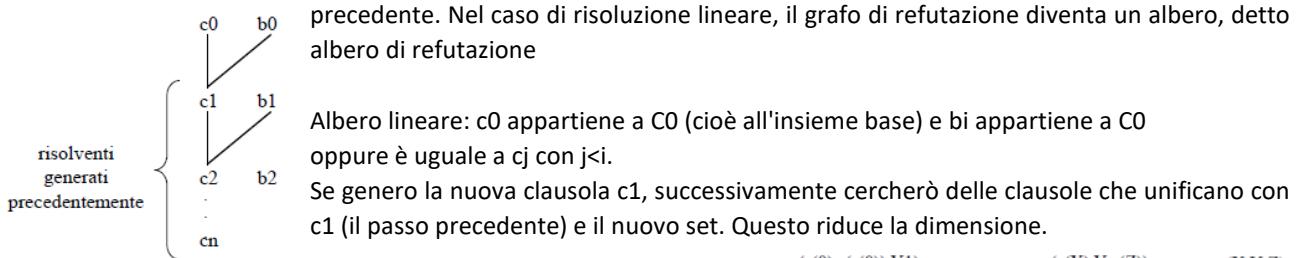
$$\begin{aligned} H &= HC = \{\text{on}(b1,\text{tavola}), & (1) \\ &\text{on}(b2,\text{tavola}), & (2) \\ &\text{colore}(b1,\text{rosso}) \vee \text{colore}(b2,\text{rosso}) & (3) \\ F &= \{ \exists X (\text{on}(X,\text{tavola}) \wedge \text{colore}(X,\text{rosso})) \} \\ FC &= \{ \neg\text{on}(X,\text{tavola}) \vee \neg\text{colore}(X,\text{rosso}) \} \end{aligned}$$

Guardando i primi due rami in alto, $\text{on}(b1,\text{tavola})$ e $\neg\text{on}(X,\text{tavola})$ vengono unificati generando una nuova clausola in cui rimane solo $\text{on}(\text{colore}(X,\text{rosso}))$, ma applicando x a $b1$, mi viene $\neg\text{colore}(b1,\text{rosso})$.



9.9.1 - STRATEGIA LINEARE

Strategia lineare (completa) sceglie almeno una clausola “parent” dall’insieme base C0 oppure tra i risolventi generati precedentemente. La seconda clausola parent è sempre il risolvente ottenuto al passo



Es.

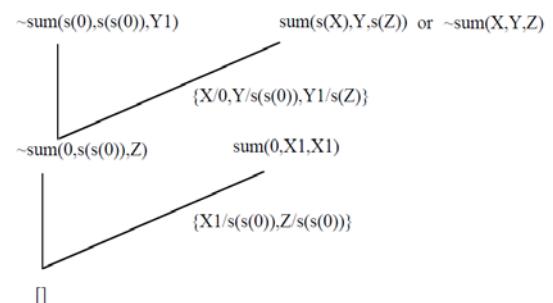
Il set c_0 in questo caso cresce dinamicamente, invece sarebbe molto meglio se quel set fosse sempre fisso.

Si può dimostrare che se io non aggiungo quello che trovo a c_0 , non ho problemi.

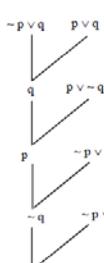
Se pensiamo all’esempio di slide 35 ho $\text{not}(q)$ e lo confronto con q dicendo “termino”, però q non era nel set di c_0 : l’ho aggiunto perché nella strategia lineare devo aggiungere quelli che trovo.

Se non lo aggiungessi, non troverei la soluzione che invece c’è, cioè la strategia diventerebbe non completa.

Il problema allora è: fino a che punto parliamo di efficienza e fino a che punto parliamo di completezza? Siamo disposti a rinunciare all’efficienza in favore della completezza?



9.9.2 - STRATEGIA LINEAR-INPUT



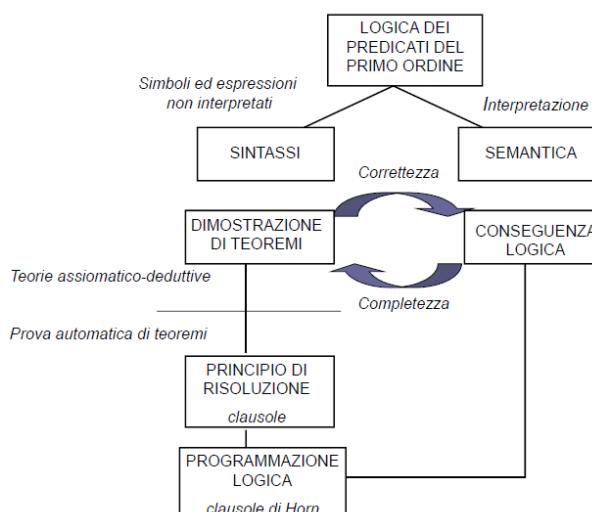
Questa strategia non è completa perché sceglie di avere sempre una clausola “parent” nell’insieme base C_0 , mentre la seconda clausola “parent” è il risolvente derivato al passo precedente.

Questa strategia non accresce mai il set di ingresso, C_0 rimane fisso e ad ogni passo prendiamo il goal e guardiamo se nelle clausole di partenza c’è la possibilità di arrivare alla soluzione.

Caso particolare della risoluzione lineare: PRO: memorizzare solo l’ultimo risolvente, CONTRO: non completa

Es.

.....
 $C_0 = \{\text{pvq}, \text{~pvq}, \text{pv} \sim q, \text{~pv} \sim q\}$ insoddisfacibile, ma la risoluzione con strategia “linear-input” produce sempre clausole che hanno almeno un letterale, e quindi non è in grado di derivare la clausola vuota. Se ci si limita, però, a un sottoinsieme delle clausole, in particolare alle clausole di Horn, allora la strategia “linear-input” è completa.



9.10 - CLAUSOLE DI HORN

La logica a clausole di Horn è un sottoinsieme della logica a clausole. Le clausole di Horn hanno al più un letterale positivo. Le clausole possono essere scritte in una forma equivalente sostituendo al simbolo di disgiunzione \vee e negazione \sim il simbolo di implicazione logica (\rightarrow) ricordando che: $\sim B \vee A$ equivale a $B \rightarrow A$

Nel seguito si indicherà l'implicazione logica $B \rightarrow A$ con: $A \leftarrow B$ ("B implica A", oppure "A se B").

La clausola: $A_1 \vee \dots \vee A_n \sim B_1 \vee \dots \vee B_m$ può essere riscritta come: $A_1, \dots, A_n \leftarrow B_1, \dots, B_m$ dove i simboli $,$ che separano gli atomi A_i sono da interpretare come disgiunzioni, mentre quelli che separano gli atomi B_j sono congiunzioni. Clausole di Horn:

$$A \leftarrow B_1, \dots, B_m \quad \leftarrow B_1, \dots, B_m \text{ Goal}$$

Molte formule della logica dei predicati possono essere scritte come clausole di Horn, questo grazie alla loro grande potenza espressiva.

Es. $\forall x \text{ gatto}(x) \vee \text{cane}(x) \rightarrow \text{animale}(x) \Rightarrow \text{animale}(x) \leftarrow \text{gatto}(x), \text{animale}(x) \leftarrow \text{cane}(x)$

9.11 – GMP: MODUS PONENS GENERALIZZATO

$$\frac{p_1', p_2', \dots, p_n', (p_1 \wedge p_2 \wedge \dots \wedge p_n \Rightarrow q)}{q} \text{ dove } p_i' \theta = p_i \theta \text{ per ogni } i$$

È utilizzato con KB di clausole definite (cioè che hanno esattamente un letterale positivo). Tutte le variabili sono universalmente quantificate e ciò corrisponde alla risoluzione con strategia linear input.

Es. La legge dice che è un crimine per un Americano vendere armi a nazioni ostili. Il paese Nono, un nemico dell'America, ha qualche missile e tutti i suoi missili gli sono stati venduti dal Colonnello West, che è americano. Dimostra che il Colonnello West è un criminale.

(Nella traduzione useremo una diversa (inversa) notazione(Russel-Norvig): Variabili x,y,z ecc; Costanti con una maiuscola.)

KB:

..è un crimine per un Americano vendere armi a nazioni ostili:

$\text{American}(x) \wedge \text{Weapon}(y) \wedge \text{Sells}(x,y,z) \wedge \text{Hostile}(z) \Rightarrow \text{Criminal}(x)$

Nono ha qualche missile: $\exists x \text{ Owns}(\text{Nono},x) \wedge \text{Missile}(x) : \text{Owns}(\text{Nono},M1) \text{ and } \text{Missile}(M1)$

Tutti i suoi missili gli sono stati dati dal Col. West: $\text{Missile}(x) \wedge \text{Owns}(\text{Nono},x) \Rightarrow \text{Sells}(\text{West},x,\text{Nono})$

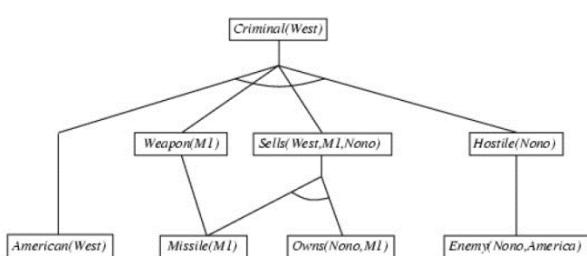
I missili sono armi: $\text{Missile}(x) \Rightarrow \text{Weapon}(x)$

Un nemico Americano è considerato ostile: $\text{Enemy}(x,\text{America}) \Rightarrow \text{Hostile}(x)$

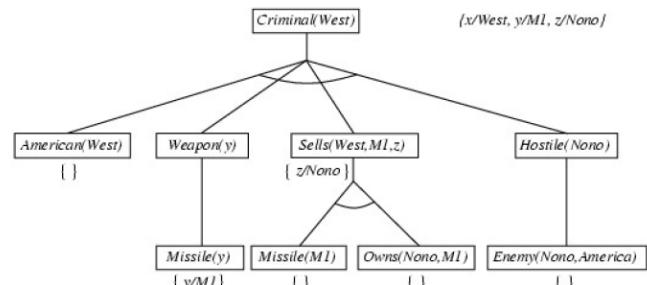
West è americano: $\text{American}(\text{West})$

Il paese Nono è un nemico dell'America: $\text{Enemy}(\text{Nono},\text{America})$

FORWARD CHAINING: dal basso verso l'alto



BACKWARD CHAINING: dall'alto verso il basso



10 - PROLOG

10.1 - INTRODUZIONE AL PROLOG

PROLOG: PROgramming in LOGic. Si fonda sulle idee di Programmazione Logica avanzate da Kowalski ed è **basato sulla logica dei Predicati del Primo Ordine** (prova automatica di teoremi - risoluzione).

È un **manipolatore di Simboli** (e non di numeri) ed è un linguaggio ad altissimo livello. Si usa in molte applicazioni AI e si basa sul **paradigma dichiarativo**. Lavora su strutture ad albero, anche i programmi sono strutture dati manipolabili e si utilizza la ricorsione (e non l'assegnamento). La metodologia di programmazione prevede di **concentrarsi sulla specifica del problema rispetto alla strategia di soluzione**.

Svantaggi:

- Linguaggio relativamente giovane
- Efficienza paragonabile a quella del LISP
- Non adatto ad applicazioni numeriche o in tempo reale
- Mancanza di ambienti di programmazione evoluti

La conoscenza sul problema è indipendente dal suo utilizzo perché mi basta esprimere COSA e non COME. Il Prolog ha un'alta modularità e flessibilità ed è lo schema progettuale alla base di gran parte dei sistemi basati sulla conoscenza (detti anche **sistemi esperti**).

È il più noto linguaggio di Programmazione Logica: il suo motto è **ALGORITMO = LOGICA + CONTROLLO**.

La parte di Logica ci serve per la conoscenza sul problema e per quanto riguarda correttezza ed efficienza.

La parte di Controllo riguarda la strategia risolutiva e si valuta in termini di efficienza.

Un programma Prolog è un insieme di clausole di Horn che rappresentano:

- **FATTI** o **ASSERZIONI** riguardanti gli oggetti in esame e le relazioni che intercorrono
Forma (cl1) A. (con A formula atomica)
- **REGOLE** sugli oggetti e sulle relazioni (Se... Allora)
Forma (cl2) A :- B1, B2, ..., Bn (con A e Bi formule atomiche)
- **GOAL** (clausole senza testa), sulla base della conoscenza definita
Forma (cl3) :- B1, B2, ..., Bn (con A e Bi formule atomiche)

Un **goal** viene provato provando i **singoli letterali da sinistra a destra**

:- collega(X,Y), persona(X), persona(Y).

Un **goal atomico** (ossia formato da un singolo letterale) viene provato confrontandolo e unificandolo con le teste delle clausole contenute nel programma.

Se esiste una sostituzione per cui il confronto ha successo:

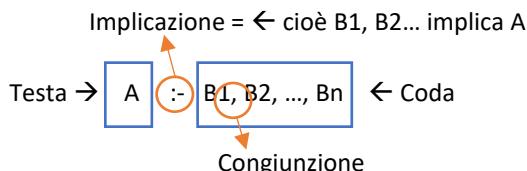
- Se la clausola con cui unifica è un fatto, la prova termina
- Se la clausola con cui unifica è una regola, ne viene provato il Body

Se non esiste una sostituzione il goal fallisce.

Es.
append([],X,X).
append([X|Z],Y,[X|T]) :- append(X,Y,T).
:- append([a,b],[c,d],[a,b,c,d]).

Viene quindi provato il body dopo aver effettuato le sostituzioni :- append([], [c,d], [a,b,c,d]).

Questo goal atomico viene provato unificandolo con la testa della prima regola che è un fatto e quindi la prova termina con successo.



Una formula atomica è una formula del tipo p(t1,t2,...,tn) in cui p è un simbolo predicativo e t1,t2,...,tn sono termini.

Un termine è definito ricorsivamente come segue:

- **Le costanti** (numeri interi/floating point, stringhe alfanumeriche aventi come primo carattere una lettera minuscola) sono termini
- **Le variabili** (stringhe alfanumeriche aventi come primo carattere una lettera maiuscola oppure il carattere “_”) sono termini.
- **$f(t_1, t_2, \dots, t_k)$ è un termine se “ f ” è un simbolo di funzione (operatore) a k argomenti e t_1, t_2, \dots, t_k sono termini.**
 $f(t_1, t_2, \dots, t_k)$ viene detta **struttura**

NOTA: le costanti possono essere viste come simboli funzionali a zero argomenti.

10.1.1 - INTERPRETAZIONE DICHIARATIVA

Le variabili all'interno di una clausola sono quantificate universalmente.

Per ogni fatto $p(t_1, t_2, \dots, t_m)$. Se X_1, X_2, \dots, X_n sono le variabili che compaiono in t_1, t_2, \dots, t_m , il significato è: $\forall X_1, \forall X_2, \dots, \forall X_n (p(t_1, t_2, \dots, t_m))$.

Per ogni regola del tipo $A:- B_1, B_2, \dots, B_k, \text{ se } Y_1, Y_2, \dots, Y_n$ sono le variabili che compaiono solo nel body della regola e X_1, X_2, \dots, X_n sono le variabili che compaiono nella testa e nel corpo, il significato è:

$\forall X_1, \forall X_2, \dots, \forall X_n, \forall Y_1, \forall Y_2, \dots, \forall Y_n ((B_1, B_2, \dots, B_k) \rightarrow A)$

$\forall X_1, \forall X_2, \dots, \forall X_n (\exists Y_1, \exists Y_2, \dots, \exists Y_n (B_1, B_2, \dots, B_k) \rightarrow A)$

Es. $\text{padre}(X, Y)$ “ X è il padre di Y ”

$\text{madre}(X, Y)$ “ X è la madre di Y ”

$\text{nonno}(X, Y) :- \text{padre}(X, Z), \text{padre}(Z, Y).$ = “ $\forall X, Y, X$ è il nonno di Y se $\exists Z$ tale che X è padre di Z e Z è il padre di Y ”

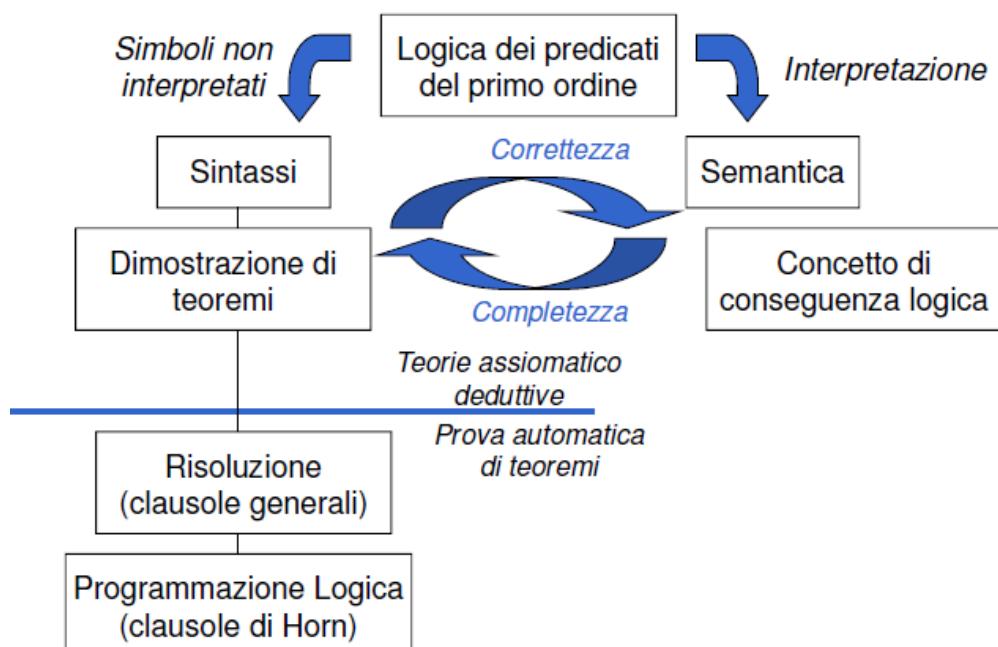
$\text{nonno}(X, Y) :- \text{padre}(X, Z), \text{madre}(Z, Y).$ = “ $\forall X, Y, X$ è il nonno di Y se $\exists Z$ tale che X è padre di Z e Z è la madre di Y ”

Una computazione corrisponde al tentativo di dimostrare, tramite la risoluzione, che una formula segue logicamente da un programma (è un teorema).

Inoltre, si deve determinare una sostituzione per le variabili del goal (detto anche “query”) per cui la query segue logicamente dal programma.

Dato un programma P e la query: $:- p(t_1, t_2, \dots, t_m)$, se X_1, X_2, \dots, X_n sono le variabili che compaiono in t_1, t_2, \dots, t_m , ha significato: $\exists X_1, \exists X_2, \dots, \exists X_n p(t_1, t_2, \dots, t_m)$ e l'obiettivo è quello di trovare una sostituzione

$\sigma = \{X_1/s_1, X_2/s_2, \dots, X_n/s_n\}$ dove si sono termini tale per cui $P \models [p(t_1, t_2, \dots, t_m)] \sigma$



10.1.2 – RISOLUZIONE SLD

Si considerano **solo clausole di Horn** (che hanno al più un letterale positivo) **in cui il letterale positivo corrisponde alla testa della clausola**. Si adotta una strategia risolutiva particolarmente efficiente che è la **Risoluzione SLD** (**corrisponde al Backward chaining per clausole di Horn**). Essa è non completa per la logica a clausole, ma completa per il sottoinsieme delle clausole di Horn.

Dato un programma logico P e una clausola goal G₀, ad ogni passo di risoluzione si ricava un nuovo risolvente G_{i+1}, se esiste, dalla clausola goal ottenuta al passo precedente G_i e da una variante di una clausola appartenente a P. Una variante per una clausola C è la clausola C' ottenuta da C rinominando le sue variabili (renaming).

La Risoluzione SLD seleziona un atomo A_m dal goal G_i secondo un determinato criterio, e lo unifica se possibile con la testa della clausola C_i attraverso la sostituzione più generale: Most General Unifier (MGU) θ_i.

Il nuovo risolvente è ottenuto da G_i riscrivendo l'atomo selezionato con la parte destra della clausola C_i ed applicando la sostituzione θ_i.

Più in dettaglio

:- A ₁ , ..., A _{m-1} , A _m , A _{m+1} , ..., A _k .	Risolvente
A:- B ₁ , ..., B _q .	Clausola del programma P
e[A _m]θ _i = [A]θ _i	allora la risoluzione SLD deriva il nuovo risolvente
:- [A ₁ , ..., A _{m-1} , B ₁ , ..., B _q , A _{m+1} , ..., A _k]θ _i .	

10.1.3 - UNIFICAZIONE

L'unificazione è un meccanismo che permette di calcolare una sostituzione al fine di rendere uguali due espressioni. Per espressione intendiamo un termine, un letterale o una congiunzione/disgiunzione di letterali.

Sostituzione: θ=[X₁/T₁, X₂/T₂, ..., X_n/T_n] insieme di legami di termini T_i a variabili X_i che rendono uguali due espressioni. L'applicazione di una sostituzione a un'espressione E, [E]θ produce una nuova espressione in cui vengono sostituite tutte le variabili di E con i corrispondenti termini.

Es.
 Espressione 1: c(X,Y)
 Espressione 2: c(a,K)
 Sostituzione unificatrice: θ = [X/a, Y/K]

COMPOSIZIONE DI SOSTITUZIONI

La composizione di sostituzioni θ₁θ₂ con θ₁=[X₁/T₁, X₂/T₂, ..., X_n/T_n] e θ₂=[Y₁/Q₁, Y₂/Q₂, ..., Y_n/Q_n] è:

θ₁θ₂=[X₁/[T₁]θ₂, ..., X_n/[T_n]θ₂, Y₁/Q₁, Y₂/Q₂, ..., Y_n/Q_n], equivale quindi ad applicare prima θ₁ e poi θ₂.

Due atomi A₁ e A₂ sono unificabili se esiste una sostituzione θ tale che [A₁]θ = [A₂]θ.

Es. θ₁=[X/f(Z), W/R, S/c], θ₂=[Y/X, R/W, Z/b] → θ₁θ₂=[X/f(b), S/c, Y/X, R/W, Z/b]

Una sostituzione θ₁ è più generale di un'altra θ₂ se esiste una terza sostituzione θ₃ tale che θ₂ = θ₁θ₃. Esistono in generale più sostituzioni unificatrici. Noi siamo interessati alla MGU. Esiste un algoritmo che calcola l'unificazione più generale se due atomi sono unificabili, altrimenti termina in tempo finito nel caso in cui i due atomi non sono unificabili.

		T2	costante c ₂	variabile X ₂	termine composto S ₂	
		T1	costante c ₁	costante c ₂	variabile X ₂	termine composto S ₂
costante c ₁	T2	unificano se c ₁ =c ₂	unificano X ₂ =c ₁	non unificano		
	X1	unificano X ₁ =c ₂	unificano X ₁ =X ₂	unificano X ₁ =S ₂		
	S1	non unificano	unificano X ₂ =S ₁	Unificano se uguale funtore e parametri unificabili		

10.1.4 - OCCUR CHECK

L'unificazione tra una variabile X e un termine composto S è molto delicata: infatti è importante controllare che il termine composto S non contenga la variabile da unificare X. Questo inficerrebbe sia la terminazione, sia la correttezza dell'algoritmo di unificazione.

Es. Si consideri l'unificazione tra $p(X,X)$ e $p(Y,f(Y))$. La sostituzione è $[X/Y, X/f(X)]$. Chiaramente, due termini unificati con lo stesso termine, sono uguali tra loro. Quindi, $Y/f(Y)$, ma questo implica $Y=f(f(f(f(\dots))))$ e il procedimento non termina.

10.1.5 - DERIVAZIONE SLD

Una derivazione SLD per un goal G0 dall'insieme di clausole definite P, è una sequenza di clausole goal G0,...Gn, una sequenza di varianti di clausole del programma C1, ...Cn, e una sequenza di sostituzioni MGU θ1, ..., θn tali che Gi+1 è derivato da Gi e da Ci+1 attraverso la sostituzione θn. La sequenza può essere anche infinita.

Esistono tre tipi di derivazioni;

- **Successo:** se per n finito Gn è uguale alla clausola vuota Gn = :-
- **Fallimento finito:** se per n finito non è più possibile derivare un nuovo risolvente da Gn e Gn non è uguale a :-
- **Fallimento infinito:** se è sempre possibile derivare nuovi risolventi tutti diversi dalla clausola vuota

Es. Derivazione di successo

sum(0, X, X). (CL1)
sum(s(X), Y, s(Z)) :- sum(X, Y, Z). (CL2)

Goal G0 :- sum(s(0), 0, W) ha una derivazione di successo perché

C1: variante di CL2 sum(s(X1), Y1, s(Z1)) :- sum(X1, Y1, Z1).

θ1 = [X1/0, Y1/0, W/s(Z1)]

G1 = :- sum(0, 0, Z1).

C2: variante di CL1 sum(0, X2, X2).

θ2 = [Z1/0, X2/0]

G2 = :-

Es. Derivazione di fallimento finita

sum(0, X, X). (CL1)
sum(s(X), Y, s(Z)) :- sum(X, Y, Z). (CL2)

Goal G0 :- sum(s(0), 0, 0) ha una derivazione di fallimento finito perché l'unico atomo del goal non è unificabile con alcuna clausola del programma.

Es. Derivazione di fallimento infinita

sum(0, X, X). (CL1)
sum(s(X), Y, s(Z)) :- sum(X, Y, Z). (CL2)

Goal G0 :- sum(A, B, C) ha una derivazione SLD infinita, ottenuta applicando ripetutamente varianti della seconda clausola di P.

C1: variante di CL2 sum(s(X1), Y1, s(Z1)) :- sum(X1, Y1, Z1).

θ1 = [A/s(X1), B/Y1, C/s(Z1)]

G1 = :- sum(X1, Y1, Z1).

C2: variante di CL2 sum(s(X2), Y2, s(Z2)) :- sum(X2, Y2, Z2).

θ2 = [X1/s(X2), Y1/Y2, Z1/s(Z2)]

G2 = :- sum(X2, Y2, Z2). ...

LEGAMI PER VARIABILI IN USCITA

Risultato della computazione:

- **Eventuale successo**
- **Legami per le variabili del goal G0**, ottenuti componendo le sostituzioni MGU applicate

Se il goal G0 è del tipo $\neg A_1(t_1, \dots, t_k), A_2(t_{k+1}, \dots, t_h), \dots, A_n(t_{j+1}, \dots, t_m)$, i termini ti "ground" rappresentano i **valori di ingresso** al programma, mentre i **termini variabili** sono i destinatari dei **valori di uscita** del programma. Dato un programma logico P e un goal G0, una risposta per PU{G0} è una sostituzione per le variabili di G0.

Si consideri una refutazione SLD per $\text{PU}\{G_0\}$. Una risposta calcolata q per $\text{PU}\{G_0\}$ è la sostituzione ottenuta restringendo la composizione delle sostituzioni MGU q_1, \dots, q_n utilizzate nella refutazione SLD di $\text{PU}\{G_0\}$ alle variabili di G_0 . **La risposta calcolata o sostituzione di risposta calcolata è il “testimone” del fatto che esiste una dimostrazione costruttiva di una formula quantificata esistenzialmente (la formula goal iniziale).**

Es. $\text{sum}(0, X, X). \quad (\text{CL1})$

$\text{sum}(s(X), Y, s(Z)) :- \text{sum}(X, Y, Z). \quad (\text{CL2})$

$G = :-\text{sum}(s(0), 0, W)$ la sostituzione $\theta = \{W/s(0)\}$ è la risposta calcolata, ottenuta componendo θ_1 con θ_2 e considerando solo la sostituzione per la variabile W di G .

10.1.6 - NON DETERMINISMO

Nella risoluzione SLD così come è stata enunciata **si hanno due forme di non determinismo:**

- La prima forma di non determinismo è legata alla selezione di un atomo A_m del goal da unificare con la testa di una clausola. Risulta definendo una particolare **regola di calcolo**
- La seconda forma di non determinismo è legata alla scelta di quale clausola del programma P utilizzare in un passo di risoluzione. Risulta definendo una **strategia di ricerca**

REGOLA DI CALCOLO

Una regola di calcolo è una funzione che ha come dominio l'insieme dei goal e che seleziona un suo atomo A_m dal goal $:- A_1, \dots, A_{m-1}, A_m, A_{m+1}, \dots, A_k$, (con A_m atomo selezionato).

Es. $\text{sum}(0, X, X). \quad (\text{CL1})$

$\text{sum}(s(X), Y, s(Z)) :- \text{sum}(X, Y, Z). \quad (\text{CL2})$

$G_0 = :-\text{sum}(0, s(0), s(0)), \text{sum}(s(0), 0, s(0))$.

Se si seleziona l'atomo più a sinistra al primo passo, unificando l'atomo $\text{sum}(0, s(0), s(0))$ con la testa di CL1, si otterrà: $G_1 = :-\text{sum}(s(0), 0, s(0))$.

Se si seleziona l'atomo più a destra al primo passo, unificando l'atomo $\text{sum}(s(0), 0, s(0))$ con la testa di CL2, si avrà: $G_1 = :-\text{sum}(0, s(0), s(0)), \text{sum}(0, 0, 0)$.

La regola di calcolo influenza solo l'efficienza, non influenza né la correttezza né la completezza del dimostratore.

Proprietà (Indipendenza dalla regola di calcolo)

- Dato un programma logico P , l'insieme di successo di P non dipende dalla regola di calcolo utilizzata dalla risoluzione SLD.

STRATEGIE DI RICERCA

Definita una regola di calcolo, nella risoluzione SLD resta un ulteriore grado di non determinismo poiché possono esistere più teste di clausole unificabili con l'atomo selezionato.

Es. $\text{sum}(0, X, X). \quad (\text{CL1})$

$\text{sum}(s(X), Y, s(Z)) :- \text{sum}(X, Y, Z). \quad (\text{CL2})$

$G_0 = :-\text{sum}(W, 0, K)$.

Se si sceglie la clausola CL1 si ottiene il risolvente $G_1 = :-$

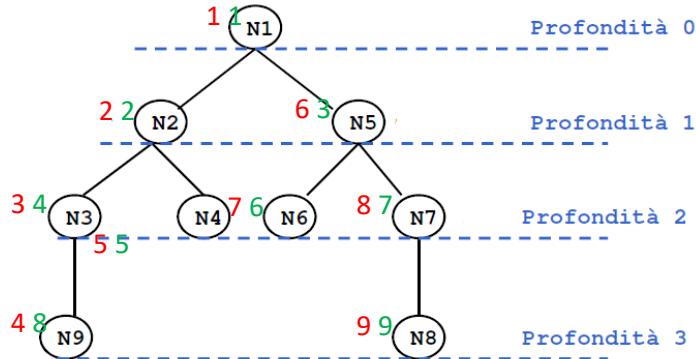
Se si sceglie la clausola CL2 si ottiene il risolvente $G_1 = :-\text{sum}(X_1, 0, Z_1)$

Questa forma di non determinismo implica che possano esistere più soluzioni alternative per uno stesso goal. La risoluzione SLD, per essere completa, deve essere in grado di generare tutte le possibili soluzioni e quindi deve considerare ad ogni passo di risoluzione tutte le possibili alternative. La strategia di ricerca deve garantire questa completezza. Una forma grafica utile per rappresentare la risoluzione SLD e questa forma di non determinismo sono gli alberi SLD.

La strategia di ricerca stabilisce perciò una particolare modalità di esplorazione dell'albero SLD alla ricerca dei rami di successo. Le modalità di esplorazione dell'albero più comuni sono Depth first e Breadth first. Entrambe le modalità implicano l'esistenza di un meccanismo di backtracking per esplorare tutte le strade alternative che corrispondono ai diversi nodi dell'albero.

Strategia Depth-First: ricerca in profondità, vengono prima esplorati i nodi a profondità maggiore. **Non completa.** (n)

Strategia Breadth-First: ricerca in ampiezza, vengono prima esplorati i nodi a profondità minore. **Completa.** (n)



Nel caso degli alberi SLD, lo spazio di ricerca non è esplicito, ma resta definito implicitamente dal programma P e dal goal G0. I nodi corrispondono ai risolventi generati durante i passi di risoluzione. I figli di un risolvente Gi sono tutti i possibili risolventi ottenuti unificando un atomo A di Gi, selezionato secondo una opportuna regola di calcolo, con le clausole del programma P. Il numero di figli generati corrisponde al numero di clausole alternative del programma P che possono unificare con A. Agli alberi SLD possono essere applicate entrambe le strategie discusse in precedenza.

Nel caso di alberi SLD, attivare il “backtracking” implica che tutti i legami per le variabili determinati dal punto di “backtracking” in poi non devono essere più considerati.

10.1.7 - ALBERI SLD

Dato un programma logico P, un goal G0 e una regola di calcolo R, un albero SLD per $P \cup \{G0\}$ via R è definito come segue:

- Ciascun nodo dell'albero è un goal (eventualmente vuoto);
- La radice dell'albero è il goal G0;
- Dato il nodo $\text{:- } A_1, \dots, A_{m-1}, A_m, A_{m+1}, \dots, A_k$, se A_m è l'atomo selezionato dalla regola di calcolo R, allora questo nodo (genitore) ha un nodo figlio per ciascuna clausola $C_i = A_i :- B_1, \dots, B_q$ di P tale che A e A_m sono unificabili attraverso una sostituzione unificatrice più generale θ .
- Il nodo figlio è etichettato con la clausola goal $\text{:- } [A_1, \dots, A_{m-1}, B_1, \dots, B_q, A_{m+1}, \dots, A_k] \theta$ e il ramo dal nodo padre al figlio è etichettato dalla sostituzione θ e dalla clausola selezionata C_i ;
- Il nodo vuoto (indicato con “:-”) non ha figli.

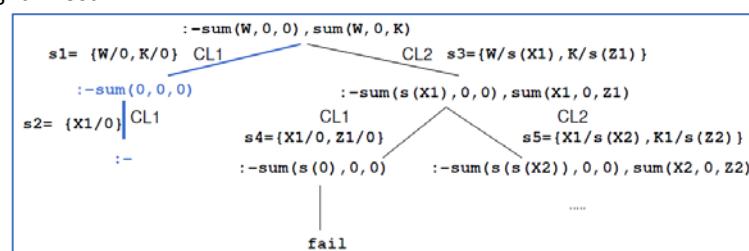
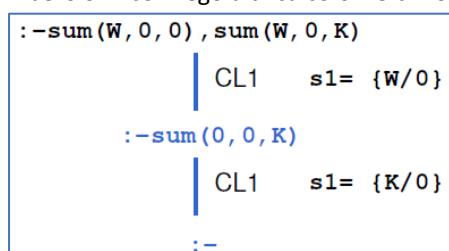
A ciascun nodo dell'albero può essere associata una profondità: la radice dell'albero ha profondità 0, mentre la profondità di ogni altro nodo è quella del suo genitore più 1.

Ad ogni ramo di un albero SLD corrisponde una derivazione SLD: ogni ramo che termina con il nodo vuoto (:-) rappresenta una derivazione SLD di successo.

La regola di calcolo influisce sulla struttura dell'albero per quanto riguarda sia l'ampiezza sia la profondità, non influisce su correttezza e completezza. Quindi, qualunque sia R, il numero di cammini di successo (se in numero finito) è lo stesso in tutti gli alberi SLD costruibili per $P \cup \{G0\}$. R influenza solo il numero di cammini di fallimento (finiti ed infiniti).

Es. $\text{sum}(0, X, X).$ (CL1)
 $\text{sum}(s(X), Y, s(Z)) :- \text{sum}(X, Y, Z).$ (CL2)
 $G0 = :- \text{sum}(W, 0, 0), \text{sum}(W, 0, K).$

Albero SLD con regola di calcolo “left-most” vs “right-most”:



Confronto albero SLD con regola di calcolo left most e right most:

- In entrambi gli alberi esiste una refutazione SLD, cioè un cammino (ramo) di successo il cui nodo finale è etichettato con ":-"
- La composizione delle sostituzioni applicate lungo tale cammino genera la sostituzione di risposta calcolata {W/0,K/0}
- Si noti la **differenza di struttura** dei due alberi. In particolare cambiano i rami di fallimento (finito e infinito)

Es. left most

```

genitore(a,b).          (R1)
genitore(b,c).          (R2)
antenato(X,Z):-genitore(X,Z) (R3)
antenato(X,Z):-genitore(X,Y),antenato(Y,Z) (R4)
G0 :- antenato(a,c)

```

10.1.8 – PROLOG

DERIVAZIONE

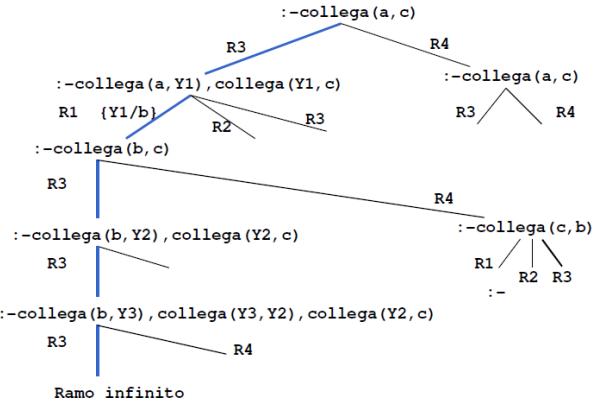
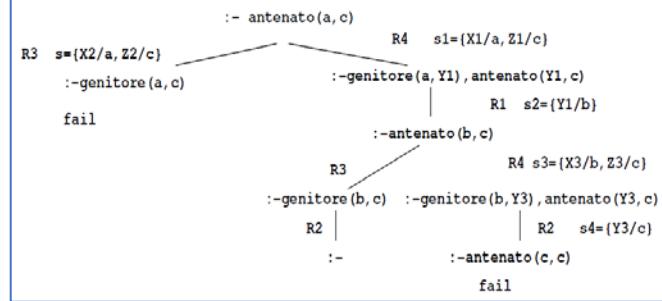
Il linguaggio Prolog, adotta la **strategia in profondità con "backtracking"** perché può essere realizzata in modo efficiente attraverso un **unico stack di goal**. Tale **stack rappresenta il ramo che si sta esplorando** e contiene opportuni riferimenti a rami alternativi da esplorare in caso di fallimento. Per quanto riguarda la scelta fra nodi fratelli, la strategia Prolog li ordina seguendo l'**ordine testuale delle clausole** che li hanno generati. La **strategia di ricerca adottata in Prolog è dunque non completa**.

```

Es. collega(a,b).          (R1)
collega(c,b).              (R2)
collega(X,Z):-collega(X,Y),collega(Y,Z). (R3)
collega(X,Y):-collega(Y,X). (R4)
Goal: :-collega(a,c)       (G0)

```

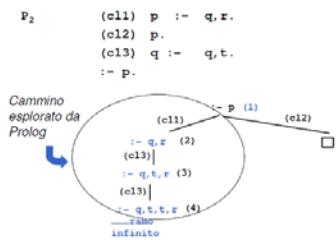
La formula collega(a,c) segue logicamente dagli assiomi, ma la procedura di dimostrazione non completa come quella che adotta la strategia in profondità non è in grado di dimostrarlo.



RISOLUZIONE

Ricerca in profondità con backtracking cronologico dell'albero di dimostrazione SLD.

Dato un letterale **G1** da risolvere, viene selezionata la **prima clausola** (secondo l'ordine delle clausole nel programma P)



la cui testa è unificabile con G1. Nel caso vi siano più clausole la cui testa è unificabile con G1, la risoluzione di G1 viene considerata come un **punto di scelta** (choice point) nella dimostrazione. In caso di fallimento in un passo di dimostrazione, Prolog ritorna in backtracking all'ultimo punto di scelta in senso cronologico (il più recente), e seleziona la clausola successiva utilizzabile in quel punto per la dimostrazione. Un problema della strategia in profondità utilizzata da Prolog è la sua **incompletezza**.

ORDINE DELLE CLAUSOLE

L'ordine delle clausole in un programma Prolog è rilevante. I due programmi P2 e P3 non sono due programmi Prolog equivalenti.

Infatti, data la "query": :-p. si ha che:

- La dimostrazione con il programma P2 non termina



- La dimostrazione con il programma **P3 ha immediatamente successo**

Una strategia di ricerca in profondità può essere realizzata in modo efficiente utilizzando tecniche non troppo differenti da quelle utilizzate nella realizzazione dei linguaggi imperativi tradizionali.

SOSTITUZIONI MULTIPLE E DISGIUNZIONE

Possono esistere più sostituzioni di risposta per una “query”. Per richiedere ulteriori soluzioni è sufficiente forzare un fallimento nel punto in cui si è determinata la soluzione che innesca il backtracking. Tale meccanismo porta ad espandere ulteriormente l’albero di dimostrazione SLD alla ricerca del prossimo cammino di successo. **In Prolog standard tali soluzioni possono essere richieste mediante l’operatore “;”.**

Il carattere ";" può essere interpretato come:

- Un operatore di disgiunzione che separa soluzioni alternative
- All’interno di un programma Prolog per esprimere la disgiunzione

INTERPRETAZIONE PROCEDURALE

Prolog può avere un’interpretazione procedurale: una procedura è un insieme di clausole di P le cui teste hanno lo stesso simbolo predicativo e lo stesso numero di argomenti (arità). Gli argomenti che compaiono nella testa della procedura possono essere visti come i parametri formali. Una “query” del tipo :- p(t1,t2,...,tn). è la chiamata della procedura p. Gli argomenti di p (ossia i termini t1,t2,...,tn) sono i parametri attuali.

L’unificazione è il meccanismo di passaggio dei parametri. Non vi è alcuna distinzione a priori tra i parametri di ingresso e i parametri di uscita (reversibilità).

Quando scriviamo un goal p(X) si può interpretare come la chiamata di una funzione procedura perché alla fine ci sarà un predicato p(x):-p(x) -> codice.

Una caratteristica tipica è il fatto che **se questa è la definizione della funzione procedura, possiamo avere più definizioni che sono i punti di scelta.**

Il corpo di una clausola può a sua volta essere visto come una sequenza di chiamate di procedure. Due clausole con le stesse teste corrispondono a due definizioni alternative del corpo di una procedura. Tutte le variabili sono a singolo assegnamento. Il loro valore è unico durante tutta la computazione e legato solo quando si cerca una soluzione alternativa (“backtracking”).

Es. `pratica_sport(mario,calcio).` `pratica_sport(giovanni,calcio).`
`pratica_sport(alberto,calcio).` `pratica_sport(marco,basket).`
`abita(mario,torino).` `abita(giovanni,genova).`
`abita(alberto,genova).` `abita(marco,torino).`

```

:- practica_sport(X,calcio). ("esiste X tale per cui X pratica il calcio?") → yes X=mario; X=giovanni; X=alberto; no
:- practica_sport(giovanni,Y). ("esiste uno sport Y praticato da giovanni?") → yes Y=calcio; no
:- practica_sport(X,Y). ("esistono X e Y tali per cui X pratica lo sport Y") → yes X=mario Y=calcio; X=giovanni Y=calcio;
X=alberto Y=calcio; X=marco Y=basket; no
:- practica_sport(X,calcio), abita(X,genova). ("esiste una persona X che pratica il calcio e abita a Genova?")
→ yes X=giovanni; X=alberto; no

```

A partire da tali relazioni, si potrebbe definire una relazione amico(X,Y) “X è amico di Y” a partire dalla seguente specifica: “X è amico di Y se X e Y praticano lo stesso sport e abitano nella stessa città”. Si noti che secondo tale relazione ogni persona è amica di se stessa.

```

amico(X,Y):- abita(X,Z)
abita(Y,Z),
pratica_sport(X,S),
pratica_sport(Y,S).
:- amico(giovanni,Y). ("esiste Y tale per cui Giovanni è amico di Y?") → yes Y = giovanni; Y=alberto; no

```

Al Prolog puro devono, tuttavia, essere aggiunte alcune caratteristiche per poter ottenere un linguaggio di programmazione utilizzabile nella pratica.

In particolare:

- **Strutture dati e operazioni** per la loro manipolazione.
- **Meccanismi per la definizione e valutazione di espressioni e funzioni.**
- **Meccanismi di input/output.**
- **Meccanismi di controllo della ricorsione e del backtracking.**
- **Negazione**

Tali caratteristiche sono state aggiunte al Prolog puro attraverso la definizione di alcuni **predicati speciali (predicati built-in)** predefiniti nel linguaggio e trattati in modo speciale dall'interprete.

10.2 – ARITMETICA E RICORSIONE

Non esiste, in logica, alcun meccanismo per la valutazione di funzioni, operazione fondamentale in un linguaggio di programmazione. I numeri interi possono essere rappresentati come termini Prolog.

Es. Il numero intero N è rappresentato dal termine: $s(s(s(\dots s(0)\dots)))$ n volte

Es. prodotto(X, Y, Z) "Z è il prodotto di X e Y"

```
prodotto(X, 0, 0).  
prodotto(X,s(Y), Z):- prodotto(X, Y, W),  
                      somma(X, W, Z).
```

Non utilizzabile in pratica: prediciati predefiniti per la valutazione di espressioni.

TABELLA OPERATORI ARITMETICI

Operatori Unari	-, exp, log, ln, sin, cos, tg
Operatori Binari	+, -, *, \, div, mod

L'insieme degli atomi Prolog contiene tanto i numeri interi quanto i numeri floating point. I principali operatori aritmetici sono simboli funzionali (operatori) predefiniti del linguaggio. In questo modo ogni espressione può essere rappresentata come un **termine Prolog**. Per gli operatori aritmetici binari il Prolog consente tanto una **notazione prefissa (funzionale)**, quanto la più tradizionale **notazione infissa**: +(2,3) e 2+3 sono due rappresentazioni equivalenti. Inoltre, 2+3*5 viene interpretata correttamente come 2+(3*5).

10.2.1 – IL PREDICATO IS

Data un'espressione, è necessario un **meccanismo per la valutazione** come il **predicato speciale predefinito is**:

Le variabili in Expr DEVONO ESSERE ISTANZIATE al momento della valutazione

T is Expr (is(T,Expr)) → T può essere un atomo numerico o una variabile, Expr deve essere un'espressione.

L'espressione Expr viene valutata e il risultato della valutazione viene unificato con T.

Es. :- X is 2+3, X is 4+1. → yes X=5

In questo caso il secondo goal della congiunzione risulta essere :- 5 is 4+1. che ha successo. X infatti è stata istanziata dalla valutazione del primo is al valore 5.

Es. :- X is 2+3, X is X+1. → no

Non corrisponde ad un assegnamento dei linguaggi imperativi. Le variabili sono write-once. Non posso fare l'assegnamento con un nuovo valore, devo proprio creare una nuova variabile che ha un altro valore. La tentazione che abbiamo di scrivere x is x+1 è alta, ma non funzionerà mai!

Nel caso dell'operatore is l'ordine dei goal cioè rilevante.

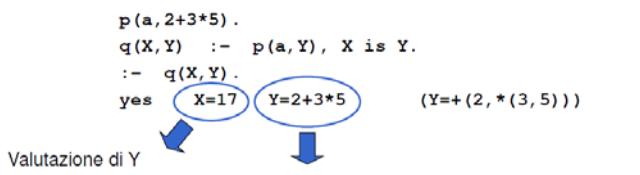
(a) :- X is 2+3, Y is X+1.

(b) :- Y is X+1, X is 2+3.

Se li invertiamo abbiamo già un fallimento perché la variabile è slegata, quindi dobbiamo stare attenti all'ordine e fare riferimento ad a, non b. Questo perché mentre il goal (a) ha successo e produce la coppia di istanziazioni X=5, Y=6, il goal (b) fallisce. Il **predicato predefinito "is"** è un tipico esempio di un predicato predefinito **non reversibile**; come conseguenza le procedure che fanno uso di tale predicato non sono (in generale) reversibili.

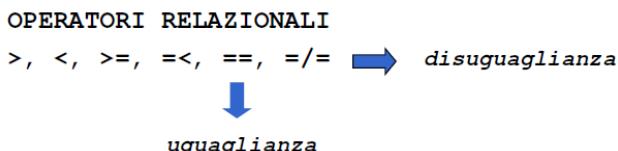
10.2.2 – TERMINI, OPERATORI ED ESPRESSIONI

Un termine che rappresenta un'espressione viene valutato solo se è il secondo argomento del predicato is.



NOTA: Y non viene valutato, ma unifica con una struttura che ha + come operatore principale, e come argomenti 2 e una struttura che ha * come operatore principale e argomenti 3 e 5

come goal all'interno di una clausola Prolog ed hanno **notazione infissa**.



Si possono anche **confrontare due espressioni** tramite l'operatore REL. I passi effettuati nella valutazione di **Expr1 REL Expr2** dove **REL** è un operatore relazionale e Expr1 e Expr2 sono

Espressioni:

- Vengono valutate Expr1 ed Expr2 (le espressioni devono essere completamente istanziate)
- I risultati della valutazione delle due espressioni vengono confrontati tramite l'operatore REL

10.2.3 – CALCOLO DI FUNZIONI

Una funzione può essere realizzata attraverso relazioni Prolog. Data una funzione f ad n argomenti, essa può essere

– $f : x_1, x_2, \dots, x_n \rightarrow y$ diventa
 $f(X_1, X_2, \dots, X_n, Y) :- <\text{calcolo di } Y>$

realizzata mediante un predicato ad n+1 argomenti nel modo seguente

Es. **Calcolare la funzione fattoriale** così definita: $fatt: n \rightarrow n!$ (n intero positivo)

Fattoriale ha input e output. Una prima cosa che deve scattare è che non definiamo funzioni, ma predicati logici, quindi degli spartiacque. I predicati sono una relazione che mettono insieme delle entità. Se devo scrivere un fattoriale, scrivo una relazione fattoriale che avrà due argomenti (uno ingresso, uno uscita).

$fatt(0) = 1$ $fatt(n) = n * fatt(n-1)$ (per $n > 0$) $fatt(0, 1)$.
 $fatt(N, Y) :- N > 0, N1 is N-1, fatt(N1, Y1), Y is N * Y1.$

Anche qui però devo scrivere in modo relazionale. Questo Y come è fatto? Vi è una condizione: solo se $N > 0$ perché se no termino direttamente. **Non possiamo usare la stessa variabile perché se ne devo modificare il valore, quindi devo creare una nuova (no assegnamenti distruttivi)** → N1 sarà il mio N-1.

Infine faccio la **ricorsione**: chiamo il fattoriale di N1 dandogli una nuova variabile di output Y1 per poter alla fine della ricorsione fare $Y = N * Y1$.

10.2.4 – RICORSIONE E ITERAZIONE

RICORSIONE TAIL

Il Prolog non fornisce alcun costrutto sintattico per l'iterazione (es costrutti while e repeat) e l'unico meccanismo per ottenere iterazione è la definizione ricorsiva. Una funzione f è definita per **ricorsione tail** se f è la funzione "più esterna" nella definizione ricorsiva o, in altri termini, se sul risultato della chiamata ricorsiva di f non vengono effettuate ulteriori operazioni. La definizione di funzioni (predicati) per ricorsione tail può essere considerata come una definizione per iterazione. Potrebbe essere valutata in spazio costante mediante un processo di valutazione iterativo.

Avere linguaggi che usano solo ricorsione ha il problema dell'efficienza di tempi di esecuzione, in quanto consuma lo stack e ogni chiamata ricorsiva accumula un record di allocazione, però ci sono delle pratiche e modalità in cui si può fare in modo che lo stack non cresca in modo così evidente.

Una delle tecniche è proprio la ricorsione tail perché fa sì che quando abbiamo una chiamata, il nuovo record di attivazione lo allochiamo direttamente su quello precedente e non abbiamo una crescita lineare sulle chiamate.

Buona prassi mettere chiamate ricorsive sempre per ultime.

Si dice **ottimizzazione della ricorsione tail** valutare una funzione tail ricorsiva f mediante un processo iterativo ossia caricando un solo record di attivazione per f sullo stack di valutazione (esecuzione). In Prolog l'ottimizzazione della ricorsione tail è complicata rispetto ai linguaggi imperativi a causa del:

- **Non determinismo**
- **Della presenza di punti di scelta** nella definizione delle clausole

Es. Vediamo un programma con tre clausole: $p(X) :- c1(X), g(X).$

(a) $p(X) :- c2(X), h1(X,Y), p(Y).$
 (b) $p(X) :- c3(X), h2(X,Y), p(Y).$

La prima non è ricorsiva, mentre le altre due lo sono e sono ricorsive tail. Supponiamo la chiamata del goal $p(Z)$ e supponiamo che allochi il $p(X)$. Il prolog ha tre/quattro stack. Vengono chiamati $c1(X)$ e $c2(X)$.

Quando crea il record di attivazione per la prima chiamata, si dovrà ricordare (ricerca breadth-first) che ci sono delle alternative e quindi quel record di attivazione e il sistema si deve ricordare che se fallisce, cioè si hanno ripercussioni sullo stack. Si dovrà ricordare anche che c'è scelta.

Due possibilità di valutazione ricorsiva del goal $:p(Z)$.

- **Se viene scelta la clausola (a)**, si deve ricordare che (b) è un punto di scelta ancora aperto. Bisogna **mantenere alcune informazioni contenute nel record di attivazione di $p(Z)$ (i punti di scelta ancora aperti)**
- **Se viene scelta la clausola (b)** (più in generale, l'ultima clausola della procedura), **non è più necessario mantenere alcuna informazione** contenuta nel record di attivazione di $p(Z)$ e la rimozione di tale record di attivazione può essere effettuata

In Prolog l'ottimizzazione della ricorsione tail è possibile solo se la scelta nella valutazione di un predicato "p" è deterministica o, meglio, se al momento della chiamata ricorsiva ($n+1$ -esima di "p" non vi sono alternative aperte per la chiamata al passo n -esimo (ossia alternative che potrebbero essere considerate in fase di backtracking). Quasi tutti gli interpreti Prolog effettuano l'ottimizzazione della ricorsione tail ed è pertanto conveniente usare il più possibile ricorsione di tipo tail.

RICORSIONE NON TAIL

Il predicato fatt è definito con una forma di ricorsione semplice (non tail). Vediamo come una **relazione ricorsiva può essere trasformata in una relazione tail ricorsiva tramite l'uso di variabili accumulatrici**.

```

fatt1(N, Y) :- fatt1(N, 1, 1, Y) .
fatt1(N, M, ACC, ACC) :- M > N.
fatt1(N, M, ACCin, ACCout) :- ACCtemp is ACCin*M,
                                M1 is M+1,
                                fatt1(N, M1, ACCtemp, Accout).

Accumulatore in ingresso   Accumulatore in uscita

```

$fatt1(N, M1, ACCtemp, Accout)$ è ricorsiva tail perché la ricorsione è alla fine.

- $fatt(2,y) \rightarrow fatt(2,1,1,y) \rightarrow$ prova con il primo: $M=1, N=2. 1>2?$ Fallisce, fa backtracking e va ad unificare con la seconda clausola. $Acctemp = Accin * M = 1 * 1 \rightarrow M1 = M+1 = 1+1 = 2$
- $fatt(2,y) \rightarrow fatt(2,1,1,y) \rightarrow fatt(2,2,1,y) \rightarrow$ prova con il primo: $M=2, N=2. 2>2?$ Fallisce, fa backtracking e va ad unificare con la seconda clausola. $Acctemp = Accin * M = 2 * 2 \rightarrow M1 = M+1 = 1+2 = 3$
- $fatt(2,y) \rightarrow fatt(2,1,1,y) \rightarrow fatt(2,2,1,y) \rightarrow fatt(2,3,2,y) \rightarrow$ prova con il primo: $M=2, N=3. 3>2?$ Sì. Y si lega ad ACC, ACC si lega a 2 e per la proprietà transitiva Y si lega a 2 e il prolog da in uscita 2.

Il fattoriale quindi viene calcolato utilizzando un argomento di accumulazione, inizializzato a 1, incrementato ad ogni passo e unificato in uscita nel caso base della ricorsione.

$$\begin{aligned}
 ACC_0 &= 1 \\
 ACC_1 &= 1 * ACC_0 = 1 * 1 \\
 ACC_2 &= 2 * ACC_1 = 2 * (1 * 1) \\
 &\dots \\
 ACC_{N-1} &= (N-1) * ACC_{N-2} = N-1 * (N-2 * (\dots * (2 * (1 * 1)) \dots)) \\
 ACC_N &= N * ACC_{N-1} = N * (N-1 * (N-2 * (\dots * (2 * (1 * 1)) \dots)))
 \end{aligned}$$

10.3 – LISTE

Le liste sono una delle strutture dati primitive più diffuse nei linguaggi di programmazione per l'elaborazione simbolica (es: Lisp). In Prolog le liste sono dei termini costruiti a partire da uno speciale atomo (che denota la lista vuota) e utilizzando un particolare operatore funzionale (l'operatore ".").

La definizione di lista può essere data ricorsivamente nel modo seguente:

- l'atomo [] rappresenta la lista vuota
- il termine . (T, Lista) è una lista se T è un termine qualsiasi e Lista una Lista. T prende il nome di TESTA della lista e Lista di CODA

Il Prolog fornisce una notazione semplificata per la rappresentazione delle liste: la lista . (T, LISTA) può essere rappresentata anche come

[T | LISTA]

Le liste di sinistra sono equivalenti e quelle nelle altre due colonne:

(1) []	[]	[]
(2) .(a, [])	[a []]	[a]
(3) .(a, .(b, []))	[a [b []]]	[a,b]
(4) .(f(g(X)), .(h(z), .(c, [])))	[f(g(X)) [h(z) [c []]]]	[f(g(X)), h(z), c]
(5) .([], [])	[[] []]	[[]]
(6) .(.(a, []), .(b, []))	[[a []] [b []]]	[[a], b]

L'unificazione (combinata con le varie notazioni per le liste) è un potente meccanismo per l'accesso alle liste (“_” significa ANY).

10.3.1 – OPERAZIONI SULLE LISTE

Le procedure che operano su liste sono definite come procedure ricorsive basate sulla definizione ricorsiva di lista.

Si possono fare diverse operazioni sulle liste:

- Verificare se un termine è una lista (is_list(T))
- Verificare se un termine appartiene ad una lista o per individuare gli elementi di una lista (member(T,L))
- Determinare la lunghezza di una lista (length(L,N))
- Concatenare due liste (append(L1,L2,L3))
- Cancellare uno o più elementi dalla lista (delete1(E1,L,L1) e delete(E1,L,L1))
- Invertire una lista (reverse(L,r) e reverse1(L,r))
- Intersecare due insiemi (intersection(S1,S2,S3))
- Unire due insiemi (union(S1,S2,S3))

is_list(T)

```
is_list(T) = true  se T è una lista      Usata per verificare se un termine è una lista.  
          false se T non è una lista  
is_list([]).      :- is_list([1,2,3]). → yes  
is_list([a|b]).   :- is_list([a|b]). → No
```

is_list([X|L]) :- is_list(L).

member(T,L)

Può essere utilizzata per la verifica di appartenenza di un elemento ad una lista o per individuare gli elementi di una lista.

```
member(T, L) "T è un elemento della lista L"  :- member(2, [1,2,3]). → yes  
                                         :- member(1, [2,3]). → no  
member(T, [T | _]).  
member(T, [_ | L])  :-  member(T, L).
```

length(L,N)

Usata per determinare la lunghezza di una lista.

length(L,N) "la lista L ha N elementi"

```
length([], 0).
length([_|L], N) :- length(L, N1),
N is N1 + 1.
```

Versione ricorsiva

```
length1(L, N) :- length1(L, 0, N).
length1([], ACC, ACC).
length1([_|L], ACC, N) :- ACC1 is ACC+1,
length1(L, ACC1, N).
```

Versione iterativa

append(L1,L2,L3)

append(L1,L2,L3) "L3 è il risultato della concatenazione di L1 e L2"

```
append([], L, L).
append([H|T], L2, [H|T1]) :- append(T, L2, T1).
```

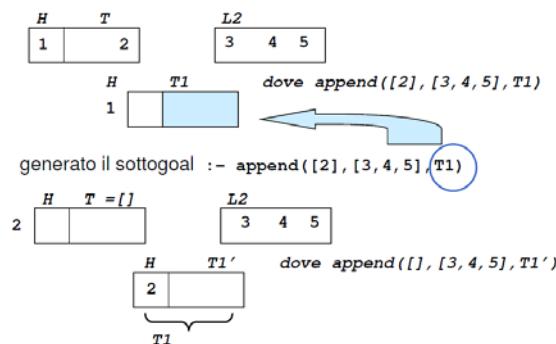
Usata per la concatenazione di due liste.

`:~ append([1,2],[3,4,5],L). → yes L = [1,2,3,4,5]`

`:~ append([1,2],L2,[1,2,4,5]). → yes L2 = [4,5]`

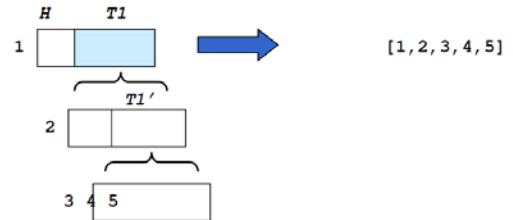
`:~ append([1,3],[2,4],[1,2,3,4]). → no`

Evoluzione della computazione in seguito alla valutazione del goal `:~ append([1,2],[3,4,5],L)`.



Viene generato il sottogoloal `:~ append([], [3, 4, 5], T1')`

Utilizzando la prima clausola si ha che `T1' = [3, 4, 5]`



delete1(E1,L,L1)

Usata per cancellare un elemento dalla lista: "la lista L1 contiene gli elementi di L tranne il primo termine unificabile con E1".

```
delete1(E1, [], []).
delete1(E1, [E1|T], T).
delete1(E1, [H|T], [H|T1]) :- delete1(E1, T, T1).
```

con E1". Se la lista è vuota, chiuso. Se la lista ha almeno un elemento, allora devo guardare se quell'elemento unifica con la testa: se sì, bene, altrimenti vuol dire che l'elemento che vogliamo cancellare non è il primo. La lista non è vuota,

quindi cade sulla seconda clausola, prova a fare l'unificazione con la testa che non funziona, allora va sulla terza clausola. Nella terza clausola toglie un elemento (H) e dirà che in uscita avrà questo elemento sempre in testa - perché non lo devo cancellare - e poi ricorsivamente chiama di nuovo la delete1. **Questo funziona se il tutto è mutuamente esclusivo**: se ho 1, 2, 3 e voglio togliere 1, in prima risposta mi dice sì e mi darà come uscita 2, 3. Cancellare 1 da 1, 2, 3 però è un punto di scelta perché unifica anche con la terza clausola, quindi Prolog deve scegliere. Prolog allora dice subito "ok, la soluzione è 2, 3" guardando la seconda clausola, però dice anche "ne vuoi anche un'altra?", fa backtracking, va nella clausola tre e usa la forma ricorsiva;

Sostanzialmente ci dice sì con una lista che contiene solo 1 ed è proprio quello che non volevamo perché noi volevamo cancellarlo. **Dov'è il problema di questa realizzazione? Io ho ipotizzato che la clausola 2 e 3 fossero mutuamente esclusive, ma per il Prolog non lo sono perché ci sono chiamate per cui entrambe possono essere applicate ad una soluzione.** Come si fa a fare la mutua esclusione? Dal punto di vista logico il sistema non sbaglia, noi le abbiamo pensate come mutuamente esclusive, ma non c'è nessuna indicazione per il programma per far sì che capisca che non deve eseguire la terza se la seconda è andata bene. **Potrei per esempio aggiungere la clausola che H deve essere diverso da E1 prima della terza clausola.** A questo punto, se io chiamo delete 1 nella lista 1, 2, 3, Prolog comunque genera un punto di scelta perché questa operazione unifica sia con la clausola 2 che con la 3. Prova con la clausola 2, a quel punto però quando piove sull'altra in backtracking, trova il predicato di diversità, questo predicato fallisce e quindi nella terza clausola non ci va.

delete(El,L,L1)

Usata per cancellare più elementi dalla lista: la lista L1 contiene gli elementi di L tranne tutti i termini unificabili con El.

```

delete(El, [], []).
delete(El, [El|T], T1) :- delete(El, T, T1).
delete(El, [H|T], [H|T1]) :- delete(El, T, T1).

```

Se lo trovo, nella testa, non continuo. Se anche in T ci fossero altri elementi, questi elementi non vengono toccati. Se voglio eliminare 2 e nella lista ho 2, 3, 2, delete unifica con 2, cioè cancella il primo che trova, ed è finita lì. La differenza con delete1 è legata a questa chiamata ricorsiva delete(El, T, T1) nella seconda clausola.

ATTENZIONE: le due procedure **delete** e **delete1** forniscono una sola risposta corretta ma non sono corrette in fase di backtracking. Come dicevamo prima, il problema è legato alla mutua esclusione tra la seconda e la terza clausola della relazione **delete**. Se T appartiene alla lista (per cui la seconda clausola di delete ha successo), allora la terza clausola non deve essere considerata una alternativa valida. Questo problema sarà risolto da **CUT**.

reverse(L,Lr)

Si usa per invertire una lista e si interpreta come: la lista Lr contiene gli elementi di L in ordine inverso. Ad esempio ho [1,

```

reverse([], []).
reverse([H|T], Lr) :- reverse(T, Tr), append(Tr, [H], Lr).

```

2, 3] e voglio ottenere [3, 2, 1]. Come realizzazione iterativa siamo tutti in grado di realizzare questo algoritmo, basta crearsi una lista accumulatrice e ci metto in testa un elemento per volta, qui invece più

che altro dobbiamo guardare la struttura: dobbiamo appendere la reverse della coda alla testa della lista Lr, poi non è finito perché c'è un caso in cui se la lista è vuota, il risultato della reverse è ovviamente una lista vuota.

Guardando la seconda clausola, la reverse di una lista che ha una testa e una coda, quindi che ha almeno un elemento, è una certa lista di uscita e questa lista di uscita è fatta dal reverse di T a cui appendo H.

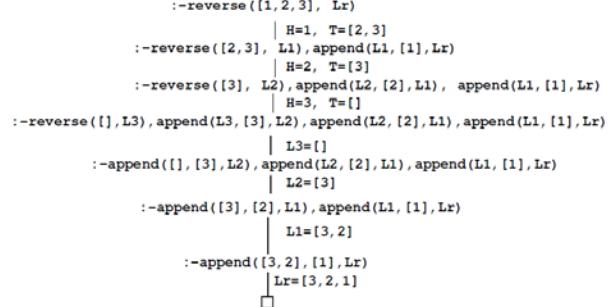
Es. 1 2 3 → 1|2=T, 3=H quindi faccio reverse di T che diventa 2|1 e questo lo appendo a 3, ottenendo 3 2 1.

Attenzione però, perché **append vuole due liste**, quindi H lo metto all'interno di una lista che contiene solo un elemento.

`:- reverse([],[]). → yes`

`:- reverse([1,2],Lr). → yes Lr = [2,1]`

Evoluzione della computazione in seguito alla valutazione del goal: questo è l'albero SLD che costruisce prolog per risolvere la reverse. Parte dal goal e fa le unificazioni, sia le ulteriori successive chiamate.



reverse1(L,Lr)

Versione iterativa di Reverse: la lista L da invertire viene diminuita ad ogni passo di un elemento e tale elemento viene

```

reverse1(L, Lr) :- reverse1(L, [], Lr).
reverse1([], ACC, ACC).
reverse1([H|T], ACC, Y) :- reverse1(T, [H|ACC], Y).

```

Potevamo usare la **append**

accumulato in una nuova lista (in testa). Un elemento viene accumulato davanti all'elemento che lo precedeva nella lista originaria, ottenendo in questo modo l'inversione. Quando la

prima lista è vuota, l'accumulatore contiene la lista invertita.

intersection(S1,S2,S3)

Gli insiemi possono essere rappresentati come liste di oggetti (senza ripetizioni) e se ne può fare l'intersezione

```

intersection([], S2, []).
intersection([H|T1], S2, [H|T3]) :- member(H, S2), intersection(T1, S2, T3).
intersection([H|T1], S2, S3) :- intersection(T1, S2, S3).

```

leggendo la regola come "l'insieme S3 contiene gli elementi appartenenti all'intersezione di S1 e S2".

Nell'intersezione devo guardare se il primo insieme è vuoto, in quel caso l'intersezione è

l'insieme vuoto, altrimenti vuol dire che nell'insieme c'è almeno un elemento e in quel caso separo la testa dal resto.

L'uscita sarà qualcosa che contiene H, se questo H appartiene al secondo elemento. Se non è member, faccio backtracking e a quel punto semplicemente butto H perché non deve appartenere all'uscita, ma ho qualcos'altro che è generato ricorsivamente. Attenzione: se appartiene, allora il terzo argomento avrà H e quell'H lo ritroviamo in uscita.

Dove c'è S3 non c'è H in testa, nella terza clausola, perché non appartiene all'intersezione.

`:- intersection([a,b], [b,c], S). → yes S=[b]`

`:- intersection([a,b,c,d], S2, [a,c]). → yes S2=[a,c | _1]`

`:- intersection(S1,S2,[a,c]). → yes S1=[a,c] S2=[a,c | _1]; S1=[a,c,_2] S2=[a,c | _1]; ... (infinite soluzioni) ...`

Nell'esempio seguente vedremo il problema della mutua esclusione tra clausole in quanto solo la prima soluzione è corretta `:- intersection([a,b,c],[b,c,d],S3). → yes S3=[b,c]; S3=[b]; S3=[c]; S3=[]; no`

`union(S1,S2,S3)`

```
union([], S2, S2).
union([X|REST], S2, S) :- member(X, S2),
    union(REST, S2, S).
union([X|REST], S2, [X|S]) :- union(REST, S2, S).
```

Nell'unione di due insiemi si legge la regola come "l'insieme S3 contiene gli elementi appartenenti all'unione di S1 e S2". Anche il predicato `union` in backtracking ha un comportamento scorretto. Infatti,

anche in questo caso non c'è mutua esclusione tra la seconda e la terza clausola.

10.4 – CUT

I predicati predefiniti consentono di influenzare e controllare il processo di esecuzione/dimostrazione di un goal. In particolare il Predicato `CUT (!)` è uno dei più importanti e complessi predicati di controllo forniti da Prolog.

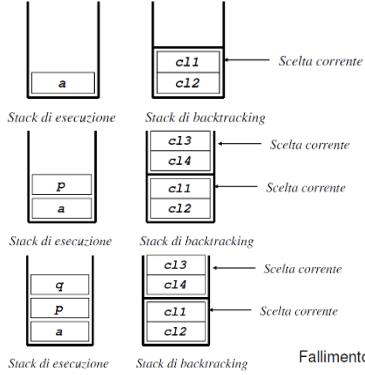
10.4.1 – MODELLO RUNTIME DI PROLOG

Nel modello run time di Prolog abbiamo due stack:

- **Stack di esecuzione** che contiene i record di attivazione delle varie procedure
- **Stack di backtracking** che contiene l'insieme dei punti di scelta. Ad ogni fase della valutazione tale stack contiene puntatori alle scelte aperte nelle fasi precedenti della dimostrazione

```
(c11)   a :- p, b.
(c12)   a :- r.
(c13)   p :- q.
(c14)   p :- r.
(c15)   r.
```

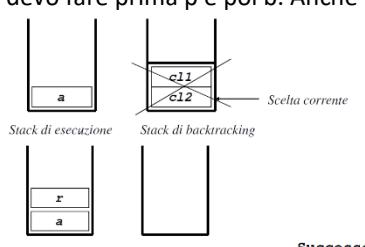
E la valutazione della query `:-a.`



Concettualmente la depth-first con backtracking funziona in questo modo: la depth-first si realizza bene con uno stack e quando genera una strada ci sono dei punti di scelta perché non espande tutto subito, ma lo espande solo se ci piove in backtracking, e così fa prolog.

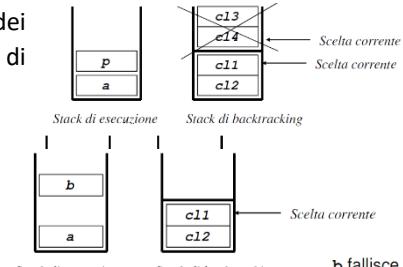
Immaginiamo che ci sia uno stack di goal da dimostrare e questo stack di goal ha come riferimento un indirizzo, che è quello delle alternative possibili, oppure accumuliamo le clausole in un altro stack e in base a quando dimostriamo i goal togliamo queste clausole dall'altro stack. Quando abbiamo una chiamata ad a e questo a viene messo nello stack del goal, è il top goal. A questo punto per a cerchiamo tutte le clausole che in qualche modo hanno un predicato, una testa, che unifica con a e qui ce ne sono due, quindi o in uno stack o con un flag o con dei puntatori bisogna ricordarsi di questo punto di scelta (questo dipende dalla realizzazione).

Proviamo `c11`, ad esempio, e se fallisce, dovrà eseguire `c12`: la scelta corrente è la prima clausola perché va in ordine testuale, però in questo caso non è risolta, quindi devo fare prima p e poi b. Anche per p si va a vedere se ci sono dei punti di scelta.



Adesso con `c13` devo eseguire `q`, quindi accumulo `q` come goal, ma per `q` non c'è alcun tipo di definizione, quindi non essendoci né un fatto né una regola, c'è un fallimento. Allora parte il backtracking: `q` viene eliminato e a questo punto torno su `p` che ha il punto di scelta aperto. `c13` non è applicabile, quindi applico la `c14` (e dopo la `c14` non ho più un punto di scelta), allora devo fare `r`.

Alla fine ripiombo su `a` perché per `a` avevo un punto di scelta aperto, che butto via, e vado su `a :- r`. `r` è semplice perché è un fatto, quindi `a` è risolto.



10.4.2 – EFFETTI DEL CUT

L'effetto del `CUT` è quello di rendere definitive alcune scelte fatte nel corso della valutazione dall'interprete Prolog. Dal punto di vista logico il `CUT` ha sempre successo quindi non cambia la definizione del nostro programma, però toglie dei punti di scelta da sotto il naso. Il prolog nell'esempio precedente costruisce dei punti di scelta aperti da rivisitare in backtracking, mentre il `CUT` toglie alcuni di quei punti di scelta. È comodo perché se lo so usare bene tolgo dei punti che

potrebbero generare errore (es. la mutua esclusione) e anche per ottimizzare il programma, il problema è che **si perde la dichiaratività**.

Si consideri la clausola: $p :- q_1, q_2, \dots, q_i, !, q_{i+1}, q_{i+2}, \dots, q_n.$

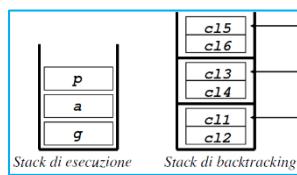
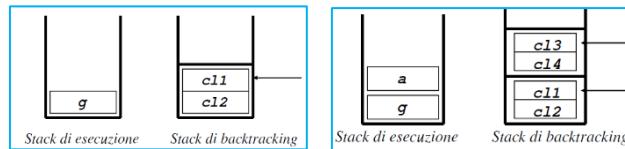
L'effetto della valutazione del goal $!$ (CUT) durante la dimostrazione del goal "p" è il seguente:

- La valutazione di $!$ ha successo (come quasi tutti i predicati predefiniti) e $!$ viene ignorato in fase di backtracking
- Tutte le scelte fatte nella valutazione dei goal q_1, q_2, \dots, q_i e in quella del goal p vengono rese definitive; in altri termini, tutti i punti di scelta per tali goal (per le istanze di tali goal utilizzate) vengono rimossi dallo stack di backtracking.
- Le alternative riguardanti i goal seguenti al CUT non vengono modificate
- **Se la valutazione di $q_{i+1}, q_{i+2}, \dots, q_n$ fallisce, fallisce tutta la valutazione di p. Infatti, anche se p o q_1, q_2, \dots, q_i avessero punti di scelta questi sarebbero eliminati dal CUT**
- Il CUT taglia rami dell'albero SLD pertanto il **CUT** non può essere definito in modo dichiarativo

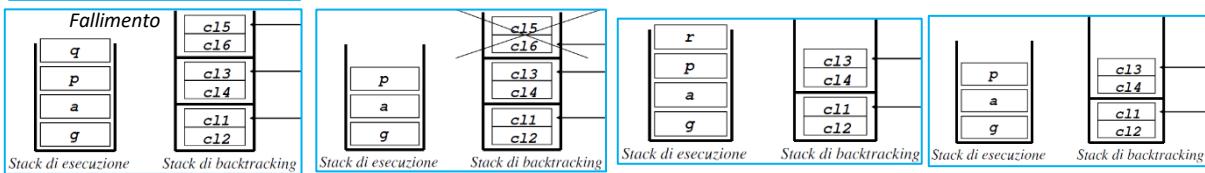
Es.

```
(c11)      g :- a.
(c12)      g :- s.
(c13)      a :- p, !, b.
(c14)      a :- r.
(c15)      p :- q.
(c16)      p :- r.
(c17)      r.!
```

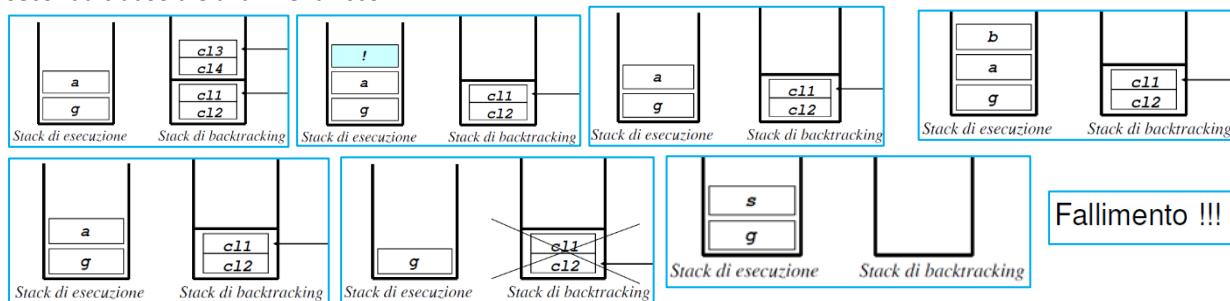
E la valutazione della query $: - g.$



p ha due punti di scelta aperti, ma essendoci il cut, cosa succede? Se proviamo a fare q, questo non ha successo, quindi facciamo r e r ha successo, allora p è risolto e viene tolto dallo stack.



Per effetto del $!$ tutti i punti di scelta per p e per a (che è il predicato in cui è inserito) sono rimossi dallo stack. p fallisce, e per a non ho più l'alternativa (che se non ci fosse questo cut avrei, invece) quindi a questo punto vado su g, eseguo la seconda clausola e alla fine fallisco.



CUT quindi si mette quando questa è l'unica giusta soluzione, l'unica giusta strada. Nell'esempio sotto, $a(X,Y)$ può unificare sia con la prima che con la seconda. Prolog prova con la prima che dice che dobbiamo fare $b(X)$, b ha o $b(1)$ o $b(2)$ e farà l'unificazione con 1, quindi $x=1$. Poi c'è il cut e questo cut taglia $b(2)$ e $a(0,0)$.

Poi va avanti su $c(Y)$ che unifica con $c(1)$ e $c(2)$, allora prolog dice sì, $x=1$ e $y=1$. Poi dice "ne vuoi un'altra?" perché ha un punto di scelta aperto su $c(2)$, quindi dice anche sì, $x=1$ e $y=2$.

Es. $a(X,Y) :- b(X), !, c(Y).$ $a(0,0).$ $b(1).$ $b(2).$ $c(1).$ $c(2).$

$:- a(X,Y).$ \rightarrow yes $X=1 Y=1;$ $X=1 Y=2;$ no

La perdita della dichiaratività è il maggiore svantaggio derivante dall'uso del CUT. Tuttavia l'uso del CUT è necessario per la correttezza di alcune classi di programmi ed è utile per l'efficienza di altre classi di programmi.

10.4.3 – MUTUA ESCLUSIONE TRA CLAUSOLE

Il cut può essere utilizzato molto semplicemente per rendere deterministica la scelta tra due o più clausole alternative.

Si supponga che la condizione $a(X)$ debba rendere le due clausole mutuamente esclusive per realizzare uno schema del tipo: if $a(.)$ then b else c . → con il CUT: $p(X) :- a(X), !, b.$ $p(X) :- c.$

Se $a(X)$ è vera, viene valutato il cut che toglie il punto di scelta per $p(X)$. Se invece $a(X)$ fallisce, si innesca il backtracking prima che il cut venga eseguito. Attenzione, la mancanza del cut rende il programma scorretto.

Es. intersezione tra insiemi

La condizione che determina la mutua esclusione è $\text{member}(H, S2)$, quindi il cut va inserito dopo tale condizione.

```
intersection([], S2, []).  
intersection([H|T], S2, [H|T3]) :- member(H, S2),  
                                intersection(T, S2, T3).  
intersection([H|T], S2, S3) :- intersection(T, S2, S3).
```



```
intersection([], S2, []).  
intersection([H|T], S2, [H|T3]) :- member(H, S2), !,  
                                intersection(T, S2, T3).  
intersection([H|T], S2, S3) :- intersection(T, S2, S3).
```

La seconda e la terza clausola ora sono mutuamente esclusive

$:- \text{intersection}([1,2,3], [2,3,4], S).$ → yes $S=[2,3];$

La presenza del "cut" rende in molti casi un programma ricorsivo deterministico e consente l'applicazione dell'ottimizzazione della ricorsione tail.

Es. Merge di due liste ordinate di numeri interi in una nuova lista ordinata.

Sebbene merge sia definita in modo tail ricorsivo, la presenza dei punti di scelta rende l'ottimizzazione

Impossibile. Inserendo il cut il programma cambia completamente in termini di efficienza.

Es. Senza il cut non è ottimizzabile:

```
merge([], L2, L2).      merge(L1, [], L1).  
merge([X|REST1],[X|REST2],[X,X|REST]) :-      merge(REST1,REST2,REST).  
merge([X|REST1],[Y|REST2],[X|REST]):-      X < Y, merge(REST1,[Y|REST2],REST).  
merge([X|REST1],[Y|REST2],[Y|REST]):-      X > Y, merge([X|REST1],REST2,REST).
```

Aggiungendo il cut diventa:

```
merge([], L2, L2).      merge(L1, [], L1).  
merge([X|REST1],[X|REST2],[X,X|REST]) :-      !, merge(REST1,REST2,REST).  
merge([X|REST1],[Y|REST2],[X|REST]):-      X < Y, !, merge(REST1,[Y|REST2],REST).  
merge([X|REST1],[Y|REST2],[Y|REST]):-      X > Y, merge([X|REST1],REST2,REST). → qui non serve
```

10.5 – LA NEGAZIONE

Finora non abbiamo preso in esame il trattamento di informazioni negative, ma solo programmi logici costituiti da clausole definite e che quindi non possono contenere atomi negati. Inoltre, attraverso la risoluzione SLD, non è possibile derivare informazioni negative.

Se vogliamo la negazione serve un sistema di rappresentazione della conoscenza più ricco: prolog è il figlio del compromesso quindi qualcosa si può fare, anche se la negazione che è disponibile in questi sistemi non è proprio la negazione logica, è un altro tipo di negazione.

Esercizio di risoluzione: è intuitivo pensare che l'affermazione persona(fido) sia falsa in quanto non dimostrabile con i fatti contenuti nel programma.

$\text{persona(maria)}.$ $\text{persona(giuseppe)}.$ $\text{persona(anna)}.$ $\text{cane(fido)}.$

Voglio dimostrare $\sim\text{persona(fido)}$ cioè not persona(fido) . Quelle sono già clausole, quindi faccio la negazione e diventa persona(fido) e poi lo aggiungo alla knowledge base e faccio la risoluzione. Siccome non trovo nulla che faccia match, non riesco a dimostrarlo.

Se invece faccio persona(fido) , lo nego e lo metto nella knowledge base, non riesco nemmeno a dimostrare il contrario. La forma di negazione di cui ci occuperemo è quella più usata e si basa su un'assunzione molto forte: la **Closed World Assumption (ipotesi di mondo chiuso)**. Se chiediamo ad un db che ha una knowledge base il "se non c'è allora è falso" diventa un "se non riesco a dimostrarlo è falso".

Se prolog dice no, sta adottando la close world assumption cioè non ha trovato una soluzione e quindi è falsa. Se chiedo a prolog " $!\text{qualcosa}$ " e il goal fallisce finitamente, il not ha successo e abbiamo una **Negation As Failure NF: dimostrazione che fallisce**.

$$\text{CWA}(P) = \{\sim A \mid \text{non esiste una refutazione SLD per } P \cup \{A\}\}$$

Questo è l'approccio seguito nelle basi di dati in cui, per problemi di dimensioni, si memorizzano solo le informazioni positive. Questo significato intuitivo è espresso dalla regola di inferenza nota come **Closed World Assumption (CWA)** o **Ipotesi di Mondo Chiuso [Reiter '78]**: Se un atomo "ground" A non è conseguenza logica di un programma P , allora si può inferire $\sim A$.

Es. $\text{capitale(roma).} \quad \text{citta}(X):-\text{capitale}(X). \quad \text{citta(bologna).}$

Con la CWA è possibile inferire $\sim \text{capitale(bologna)}$.

La CWA è un esempio di regola di inferenza NON MONOTONA in quanto l'aggiunta di nuovi assiomi alla teoria (ossia nuove clausole in un programma) può modificare l'insieme di teoremi che valevano precedentemente.

Sappiamo che i fallimenti sono di due tipi negli alberi di ricerca: finiti e infiniti. Non possiamo evitare che non ci siano fallimenti infiniti, il concetto di fallimento non è computabile completamente. Se è finito invece sì perché tutti i rami del mio albero sono rami finiti che non raggiungono la clausola vuota.

A causa dell'indecidibilità della logica del primo ordine, non esiste alcun algoritmo in grado di stabilire in un tempo finito se A non è conseguenza logica di P . Infatti, dal punto di vista operazionale, se A non è conseguenza logica di P la risoluzione SLD può non terminare. L'applicazione della CWA deve necessariamente essere ristretta agli atomi la cui prova termina in tempo finito, cioè agli atomi per cui la risoluzione SLD non diverge.

10.5.1 – NEGAZIONE PER FALLIMENTO

L'uso della CWA viene sostituito con quello di una regola meno potente, in grado di dedurre un insieme più piccolo di informazioni negative detta **Negazione per Fallimento ("Negation as Failure" - NF)** [Clark 78]. Essa si limita a derivare le negazioni di atomi la cui dimostrazione termina con fallimento in tempo finito. **Dato un programma P , se l'insieme $FF(P)$ (insieme di fallimento finito) denota gli atomi A per cui la dimostrazione fallisce in tempo finito, la regola NF si esprime come:**

$$NF(P)=\{\sim A \mid A \in FF(P)\}$$

Se un atomo A appartiene a $FF(P)$ allora A non è conseguenza logica di P , ma non è detto che tutti gli atomi che non sono conseguenza logica del programma appartengano all'insieme di fallimento finito.

$\text{citta(roma)}:-\text{citta(roma).} \quad \text{citta(bologna).}$

$citta(roma)$ non è conseguenza logica di P ma non appartiene a $FF(P)$.

10.5.2 – RISOLUZIONE SLDFN

Per risolvere goal generali, cioè che possono contenere letterali negativi, si introduce un'estensione della risoluzione SLD, nota come **risoluzione SLDFN** [Clark 78]. Combina la risoluzione SLD con la negazione per fallimento (NF).

Sia $:- L_1, \dots, L_m$ il goal (generale) corrente, in cui L_1, \dots, L_m sono letterali (atomi o negazioni di atomi). Un passo di risoluzione SLDFN si schematizza come segue:

- Non si seleziona alcun letterale negativo L_i , se non è "ground"
- Se il letterale selezionato L_i è positivo, si compie un passo ordinario di risoluzione SLD
- Se L_i è del tipo $\sim A$ (con A "ground") ed A fallisce finitamente (cioè ha un albero SLD di fallimento finito), L_i ha successo e si ottiene il nuovo risolvente $:- L_1, \dots, L_{i-1}, L_{i+1}, \dots, L_m$

Risolvere con successo un letterale negativo non introduce alcun legame (unificazione) per le variabili dal momento che si considerano solo letterali negativi "ground": al nuovo risolvente $:- L_1, \dots, L_{i-1}, L_{i+1}, \dots, L_m$ non si applica alcuna sostituzione. Una regola di calcolo si dice **SAFE** se seleziona un letterale negativo solo quando è "ground". La selezione di letterali negativi solo "ground" è necessaria per garantire correttezza e completezza della risoluzione SLDFN.

La risoluzione SLDNF è alla base della realizzazione della negazione per fallimento nei sistemi Prolog. Per dimostrare $\neg A$, dove A è un atomo, l'interprete del linguaggio cerca di costruire una dimostrazione per A. Se la dimostrazione ha successo, allora la dimostrazione di $\neg A$ fallisce, mentre se la dimostrazione per A fallisce finitamente $\neg A$ si considera dimostrato con successo.

Es. `capitale(roma).`

`capoluogo(bologna)`

`citta(X):- capitale(X).`

`citta(X):- capoluogo(X).`

`:- citta(X), ~capitale(X).`

Il linguaggio Prolog non adotta una regola di selezione safe, cioè seleziona sempre il letterale più a sinistra, senza controllare che sia "ground", che è una realizzazione non corretta della risoluzione SLDNF. Cosa succede se si seleziona un letterale negativo non ground?

Es. `capitale(roma).`

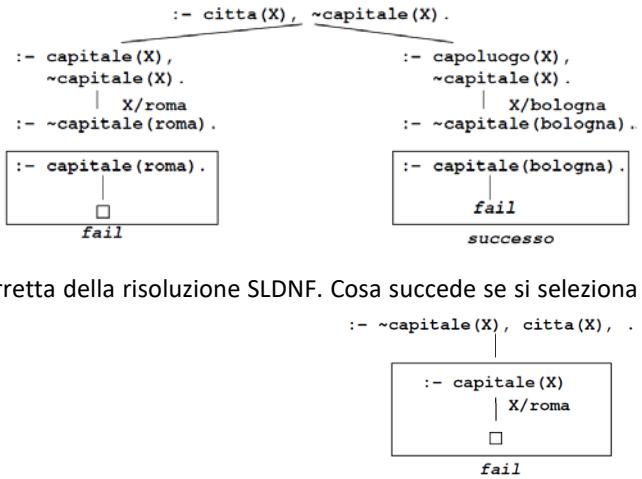
`capoluogo(bologna)`

`citta(X):- capitale(X).`

`citta(X):- capoluogo(X).`

`:- ~capitale(X), citta(X), .` → Seleziono il letterale $\neg \text{capitale}(X)$ che non è ground al momento della valutazione.

La query risponde NO ed è scorretto!



10.5.3 – NEGAZIONE E QUANTIFICATORI

Il problema della scorrettezza nasce dal fatto che **non si interpreta correttamente la quantificazione nel caso di letterali negati non "ground"**. Si consideri il programma precedente e il goal G `:- ~capitale(X).` che corrisponde alla negazione della formula $F = \exists X \neg \text{capitale}(X)$. (esiste una entità (bologna) che non è una capitale).

Con la risoluzione SLDNF, si cerca una dimostrazione per `:- capitale(X)`. ossia per $F = \exists X \text{capitale}(X)$.

Dopo di che si nega il risultato ottenendo $F = \neg (\exists X \text{capitale}(X))$. che corrisponde a $F = \forall X (\neg \text{capitale}(X))$.

Quindi, se esiste una X che è capitale, F fallisce.

10.5.4 – NEGAZIONE IN PROLOG

La forma di negazione introdotta in quasi tutti i linguaggi logici, Prolog compreso, è la negazione per fallimento. Può essere realizzata facilmente scambiando tra loro la nozione di successo e di fallimento. Il meccanismo di valutazione di una query negativa $\neg Q$ è definito in Prolog nel modo seguente: **si valuta la controparte positiva della query Q. Se Q fallisce, si ha successo, mentre se la Q ha successo la negazione fallisce.** Si noti che, data la strategia di risoluzione utilizzata dal Prolog, è possibile che la dimostrazione di Q non abbia termine (ossia **che il Prolog vada in loop in tale dimostrazione**).

Prolog adotta una strategia di selezione dei letterali left-most. Questo può generare problemi perché il significato logico di tali query è diverso da quello atteso.

Es. `:- not p(X).` Il significato di tale query è $\exists X (\text{not } p(X))$.

Prolog verifica il goal `:- p(X)` e il significato di tale query è $\exists X p(X)$

Dopo di che si nega il risultato, ossia: `not(\exists X p(X))` che corrisponde a $\forall X \text{not}(p(X))$

Mentre noi ci aspettavamo: $\exists X \text{not}(p(X))$

Riassumendo: **Prolog non adotta una regola di selezione SAFE.** Usando la regola di selezione del Prolog, si possono ottenere risultati diversi da quelli attesi a causa delle quantificazioni delle variabili. **È buona regola di programmazione verificare che i goal negativi siano sempre GROUND al momento della selezione.** Questo controllo è a carico dell'utente!

10.6 – META-PREDICATI

In Prolog **predicati (programmi)** e **termini (dati)** hanno la stessa struttura e possono essere utilizzati in modo interscambiabile.

10.6.1 – PREDICATO CALL

Un primo predicato predefinito che può essere utilizzato per trattare i dati come programmi è il predicato call. **call(T)**: il termine T viene trattato come un atomo predicativo e viene richiesta la valutazione del goal T all'interprete Prolog. Al momento della valutazione T deve essere istanziato ad un termine non numerico (eventualmente contenente variabili). Il predicato call può essere considerato come un **predicato di meta-livello** in quanto consente l'invocazione dell'interprete Prolog all'interno dell'interprete stesso. Il predicato call ha come argomento un predicato.

p(a). q(X):- p(X). :- call(q(Y)). → yes Y = a. (richiede all'interprete la dimostrazione di q(Y))

Il predicato call può essere utilizzato all'interno di programmi:

p(X):- call(X). q(a). :- p(q(Y)). → yes Y = a.

Una notazione consentita da alcuni interpreti è la seguente: p(X):- X. (Con X variabile meta-logica)

Es. Si supponga di voler realizzare un costrutto condizionale del tipo if_then_else:

if_then_else(Cond,Goal1,Goal2) → Se Cond è vera viene valutato Goal1, altrimenti Goal2

if_then_else(Cond,Goal1,Goal2):- call(Cond), !, call(Goal1).

if_then_else(Cond,Goal1,Goal2):- call(Goal2).

10.6.2 – PREDICATO FAIL

Fail è un **predicato predefinito senza argomenti**. La **valutazione del predicato fail fallisce sempre** e quindi forza l'attivazione del meccanismo di backtracking.

Si può usare il predicato fail:

- Per ottenere forme di iterazione sui dati;
- Per implementare la negazione per fallimento;
- Per realizzare una implicazione logica.

FAIL PER ITERAZIONE

Si consideri il caso in cui la **base di dati contiene una lista di fatti del tipo p/1** e si voglia **chiamare la procedura q su ogni elemento X che soddisfa il goal p(X)**. Una possibile realizzazione è la seguente:

itera :- call(p(X)), verifica(q(X)), fail.

itera.

verifica(q(X)):- call(q(X)), !.

Nota: il fail innesca il meccanismo di backtracking quindi tutte le operazioni effettuate da q(X) vengono perse, tranne quelle che hanno effetti non “backtrackabili”.

FAIL PER NEGAZIONE

Si supponga di voler realizzare il meccanismo della negazione per fallimento: not(P) è vero se P non è derivabile dal programma. Una possibile realizzazione è la seguente:

not(P) :- call(P), !, fail.

not(P).

La combinazione !,fail è interessante ogni volta si voglia, **all'interno di una delle clausole** per una relazione p, generare un fallimento globale per p (e non soltanto un backtracking verso altre clausole per p). Consideriamo il problema di voler definire una proprietà p che vale per tutti gli individui di una data classe tranne alcune eccezioni. Tipico esempio è la **proprietà vola** che vale per ogni individuo della classe degli uccelli tranne alcune eccezioni:

vola(X) :- pinguino(X), !, fail.

vola(X) :- struzzo(X), !, fail.

....

vola(X) :- uccello(X).

10.6.3 – PREDICATI SETOF E BAGOF

Ogni query `:– p(X)`. è interpretata dal Prolog in modo esistenziale; viene cioè proposta una istanza per le variabili di p che soddisfa la query. In alcuni casi può essere interessante poter rispondere a query del secondo ordine, ossia a query del tipo “quale è l’insieme S di elementi X che soddisfano la query `p(X)`?”

Molte versioni del Prolog forniscono alcuni predicati predefiniti per query del secondo ordine. I **predicati predefiniti per questo scopo sono `setof(X,P,S)`.** S è l’insieme delle istanze X che soddisfano il goal P `bagof(X,P,L)`. L è la lista delle istanze X che soddisfano il goal P.

In entrambi i casi, se non esistono X che soddisfano P i predicati falliscono:

- `bagof` produce una lista in cui possono essere contenute ripetizioni
- `setof` produce una lista corrispondente ad un insieme in cui eventuali ripetizioni sono eliminate.

Es. Supponiamo di avere un data base del tipo `p(1). p(2). p(0). p(1). q(2). r(7).`

`:– setof(X,p(X),S).` → yes `S = [0,1,2]` → `X = X`

`:– bagof(X,p(X),S).` → yes `S = [1,2,0,1]` → `X = X`

La variabile X alla fine della valutazione non è legata a nessun valore.

Es. Supponiamo di avere un data base del tipo `p(1). p(2). p(0). p(1). q(2). r(7).`

`:– setof(X, s(X),S).` → no

`:– bagof(X, s(X), S).` → no

(Questo è il comportamento atteso, ma in realtà molti interpreti danno un errore del tipo “calling an undefined procedure `s(X)`”).

Es. Supponiamo di avere un data base del tipo

`padre(giovanni,mario).` `padre(giovanni,giuseppe).`
`padre(mario, paola).` `padre(mario,aldo).`
`padre(giuseppe,maria).`

`:– setof(X, padre(X,Y), S).` → yes

<code>X=X</code>	<code>Y= aldo</code>	<code>S=[mario];</code>
<code>X=X</code>	<code>Y= giuseppe</code>	<code>S=[giovanni];</code>
<code>X=X</code>	<code>Y= maria</code>	<code>S=[giuseppe];</code>
<code>X=X</code>	<code>Y= mario</code>	<code>S=[giovanni];</code>
<code>X=X</code>	<code>Y= paola</code>	<code>S=[mario]; no</code>

Nota: non fornisce tutti gli X per cui `padre(X,Y)` è vera, ma tutti gli X per cui, per lo stesso valore di Y, `padre(X,Y)` è vera.

Per quantificare esistenzialmente Y si può usare questa sintassi:

`:– setof(X, Y^padre(X,Y), S).` → yes `[giovanni,mario,giuseppe]`

`X=X Y=Y`

`:– setof((X,Y), padre(X,Y), S).` → yes `S=[(giovanni,mario),(giovanni,giuseppe),
(mario, paola), (mario,aldo),(giuseppe,maria)]`

`X=X Y=Y`

IMPLICAZIONE CON SETOF

Setof può essere usato per realizzare un’implicazione. Es. se abbiamo predicati del tipo `padre(X,Y)` e `impiegato(Y)` e vogliamo verificare se per ogni Y `padre(p,Y) ⇒ impiegato(Y)`:

`implica(Y):- setof(X, padre(Y,X),L), verifica(L).`

`verifica([]).`

`verifica([H|T]):- impiegato(H), verifica(T).`

ITERAZIONE CON SETOF

Vediamo un esempio in cui **setof viene usato per realizzare un’iterazione**. Vogliamo chiamare la procedura q per ogni elemento per cui vale p.

`itera:- setof(X, p(X),L), scorri(L).`

`scorri([]).`

`scorri([H|T]):- call(q(H)), scorri(T).`

Nell’iterazione realizzata tramite il fail la procedura q doveva produrre effetti non rimovibili dal backtracking. In questo caso invece non è necessario.

10.6.4 – PREDICATO FINDALL

Per ottenere la stessa semantica di setof e bagof con quantificazione esistenziale per la variabile non usata nel primo argomento esiste un **predicato predefinito findall(X,P,S)** che è vero se S è la lista delle istanze X (senza ripetizioni) per cui la proprietà P è vera. Se non esiste alcun X per cui P è vera findall non fallisce, ma restituisce una lista vuota. Supponiamo di avere un data base del tipo:

```
padre(giovanni,mario).    padre(giovanni,giuseppe).      padre(mario, paola).
    padre(mario,aldo).     padre(giuseppe,maria).
:- findall(X, padre(X,Y), S). → yes S=[giovanni, mario, giuseppe]      X=X      Y=Y
Equivale a :- setof(X, Y^padre(X,Y), S).
```

I predicati setof, bagof e findall funzionano anche se le proprietà che vanno a controllare non sono definite da fatti ma da regole.

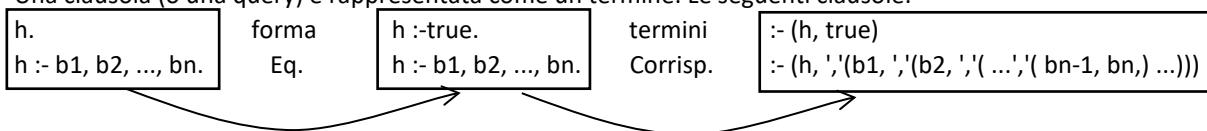
10.7 – META-INTERPRETI

In Prolog non vi è alcuna differenza sintattica tra programmi e dati e che essi possono essere usati in modo intercambiabile, ad esempio si ha la possibilità di:

- Accedere alle clausole che costituiscono un programma e trattare tali clausole come termini
- Modificare dinamicamente un programma (il data-base)
- Applicare la meta-interpretazione

10.7.1 – ACCESSO ALLE CLAUSOLE

Una clausola (o una query) è rappresentata come un termine. Le seguenti clausole:



CLAUSE(HEAD,BODY)

- Il predicato dà **vero** se **:- (HEAD,BODY)** è unificato con una clausola all'interno del data base
- Quando valutata, **HEAD** deve essere istanziata ad un termine non numerico, **BODY** può essere o una variabile o un termine che denota il corpo di una clausola.
- **Apre un punto di scelta per procedure non deterministiche** (più clausole con testa unificabile con HEAD)

Es. ?-dynamic(p/1). ?-dynamic(q/2). p(1). q(X,a) :- p(X), r(a). q(2,Y) :- d(Y).

?- clause(p(1),BODY). → yes BODY=true

?- clause(p(X),true). → yes X=1

?- clause(q(X,Y), BODY). → yes X=_1 Y=_2 BODY=p(_1), r(_2); X=2 Y=_2 BODY=d(_2); no

?- clause(HEAD,true). → Error - invalid key to data-base

10.7.2 – MODIFICHE AL DB

ASSERT(T)

Per modificare il database si può usare la **assert(T)**, che significa "la clausola T viene aggiunta al data-base". Alla valutazione, **T** deve essere istanziato ad un termine che denota una clausola (un atomo o una regola). T viene aggiunto nel data-base in una posizione non specificata. Ignorato in backtracking (non dichiarativo). Ha due varianti:

- **asserta(T)**: "la clausola T viene aggiunta all'inizio data-base"
- **assertz(T)**: "la clausola T viene aggiunta al fondo del data-base"

Es.

```
?-dynamic(a/1). → ?- assert(a(2)). → a(1). → ?- asserta(a(3)) → a(3).
a(2).           a(2).           a(1).
b(X):-a(X).    b(X):-a(X).    a(2).
                                         b(X):-a(X).
```

RETRACT(T)

retract(T) significa "la prima clausola nel data-base unificabile con T viene rimossa". Alla valutazione, **T deve essere istanziato ad un termine che denota una clausola**; se più clausole sono unificabili con T è **rimossa la prima clausola** (con punto di scelta a cui tornare in backtracking in alcune versioni del Prolog).

Alcune versioni del Prolog forniscono un secondo predicato predefinito: il predicato "abolish" (o "retract_all", a seconda delle implementazioni): abolish(NAME,ARITY).

PROBLEMI DI ASSERT E RETRACT

Utilizzare questi due predicati significa perdere la semantica dichiarativa dei programmi Prolog. Si considerino le seguenti query, in un database vuoto: `?- assert(p(a)), p(a).` `?- p(a), assert(p(a)).`

La prima valutazione ha successo, la seconda genera un fallimento. L'ordine dei letterali è rilevante nel caso in cui uno dei due letterali sia il predicato predefinito assert.

Un ulteriore problema riguarda la quantificazione delle variabili: le variabili in una clausola nel data-base sono quantificate universalmente mentre le variabili in una query sono quantificate esistenzialmente. Si consideri la query `:assert((p(X)))`. Sebbene X sia quantificata esistenzialmente, l'effetto della valutazione della query è l'aggiunta al database della clausola `p(X)`, ossia della formula $\forall X p(X)$.

10.7.3 – META-INTERPRETI

Per realizzare **meta-programmi**, ossia di programmi che operano su altri programmi, si può usare la rapida prototipazione di interpreti per linguaggi simbolici (**meta-interpreti**). In Prolog, un meta-interprete per un linguaggio L è, per definizione, un interprete per L scritto nel linguaggio Prolog.

SOLVE(GOAL)

```
solve(true) :-!.
solve((A,B)) :- !, solve(A), solve(B).
solve(A) :- clause(A,B), solve(B).
```

solve(GOAL) "il goal GOAL è deducibile dal programma Prolog puro definito da clause (ossia contenuto nel database)". Può facilmente essere esteso per trattare i predicati predefiniti del Prolog (almeno alcuni di essi). È necessario aggiungere una clausola speciale per ognuno di essi prima dell'ultima clausola per "solve".

Il meta-interprete per Prolog puro può essere modificato per adottare una regola di calcolo diversa (ad esempio rightmost).

```
solve(true) :-!.
solve((A,B)) :- !, solve(B), solve(A).
solve(A) :- clause(A,B), solve(B).
```

PRO E CONTRO

PRO	CONTRO
<ul style="list-style-type: none"> Possibilità di sfruttare a fondo le caratteristiche del Prolog Efficienza Facilità di realizzazione 	<ul style="list-style-type: none"> Approccio limitato all'uso di regole di produzione con strategia di inferenza backward Scarsa leggibilità dei programmi Scarsa modificabilità dei programmi

10.7.5 – META-INTERPRETE FORWARD

Meta-interprete che utilizza una strategia di inferenza forward. Consideriamo, in primo luogo, il caso di un interprete di base per regole rappresentate come asserzioni Prolog del tipo: rule(CONSEG,ANTEC)

Un interprete forward per le regole è definito dal seguente programma:

```
interpreta:- <verifica se si è raggiunto un obiettivo>
interpreta:- rule(CONSEG, ANTEC),
            verifica_antec(ANTEC),
            not(CONSEG),
            assert(CONSEG),
            interpreta.
interpreta :- <riporta fallimento della dimostrazione>
verifica_antec(ANTEC) "la congiunzione ANTEC è soddisfatta"
```

```

verifica_antec((A,B)):- !, call(A), verifica_antec(B).
verifica_antec(A) :- call(A).

```

Affinché una regola sia applicabile devono essere soddisfatte due condizioni:

- L'antecedente della regola deve essere soddisfatto.
- Il conseguente della regola non deve essere già vero. Ciò permette di evitare che una regola venga applicata più volte sugli stessi dati e che l'interprete vada in ciclo.

L'interprete non ha una vera e propria fase di risoluzione di conflitti: viene semplicemente attivata la prima regola applicabile. Non è difficile realizzare un meta-interprete in cui le fasi di MATCH e CONFLICT-RESOLUTION sono separate. Supponiamo che le regole siano rappresentate da asserzioni del tipo: regola(NOME, CONSEG, ANTEC) in cui NOME è un nome che identifica univocamente ogni regola.

```

interpreta1 :- <verifica se si è raggiunto un obiettivo>
interpreta1 :- match(REG_APPLICABILI),
conflict_res(REG_APPLICABILI,REGOLA),
applica(REGOLA),
interpreta1.

interpreta1 :- <riporta fallimento della dimostrazione>.
match(REG_APPLICABILI) "REG_APPLICABILI: l'insieme di regole applicabili dato il contenuto della memoria di lavoro (data base)"

match(REG_APPLICABILI) :- setof([REG,CONSEG],applic(REG,CONSEG),REG_APPLICABILI).
applic (REG,CONSEG) "la regola REG con conseguente CONSEG è applicabile"
applic(REG,CONSEG) :- regola(REG, CONSEG,ANTEC),
verifica_antec(ANTEC),
not(CONSEG).

conflict_res(REG_APPLIC,[REG,CONSEG]) "la regola REG con conseguente CONSEG è la regola selezionata all'interno della lista REG_APPLIC di regole"
conflict_res(REG_APPLIC,[REG,CONSEG]):- <selezione della regola da applicare>
applica([REG,CONSEG])
"applicazione regola REG con conseguente CONSEG"

applica([REG,CONSEG])-> assert(CONSEG).

```

PRO E CONTRO

PRO	CONTRO
<ul style="list-style-type: none"> • Elevata flessibilità • Facilità di realizzazione dei meta-interpreti • Portabilità • Leggibilità e modificabilità (almeno per meta-interpreti semplici) • Possibilità di definire meta-interpreti per diversi linguaggi di rappresentazione della conoscenza e diverse strategie di controllo 	<ul style="list-style-type: none"> • I meta-interpreti possono diventare difficili da mantenere se il linguaggio di rappresentazione e le strategie di controllo diventano molto complesse • Elevata inefficienza dovuta alla sovrapposizione di 1+ livelli di interpretazione sopra a Prolog (soluzione: tecniche di valutazione parziale)

10.8 – VINCOLI IN PROLOG

Esistono linguaggi di programmazione che combinano la dichiaratività della Programmazione Logica e l'efficienza della Risoluzione di Vincoli. Limitazioni della programmazione logica:

- Gli oggetti manipolati dai programmi logici sono strutture non interpretate per cui l'unificazione ha successo solo tra oggetti sintatticamente identici.
- Strategia di ricerca del tipo depth-first con backtracking cronologico nello spazio delle soluzioni e conducono a strategie del tipo Generate and Test.

10.8.1 – CONSTRAINT LOGIC PROGRAMMING (CLP)

Un esempio di programmazione logica a vincoli è la **Constraint Logic Programming o CLP** (1989, Jaffar Lassez).

CLP permette di:

- **Associare a ciascun oggetto la sua semantica e le operazioni primitive** che agiscono su di esso (domini di computazione quali reali, interi, razionali, booleani e domini finiti di ogni genere).
- **Sfruttare procedure di ricerca nello spazio delle soluzioni più intelligenti** che conducono ad una computazione guidata dai dati e ad uno sfruttamento attivo dei vincoli.

L'aspetto chiave della CLP è l'aumento di flessibilità derivante dall'introduzione di oggetti semanticamente primitivi su cui il linguaggio può inferire. L'unificazione è solo un caso particolare di risoluzione di vincoli. Ne deriva il superamento di una delle lacune presenti nello schema tradizionale della PL che ne inficia il meccanismo fondamentale dell'unificazione.

Due termini, intesi come strutture non interpretate, sono unificabili solo se sintatticamente identici. In tal modo la struttura 5 e la struttura 2+3 non sono considerate lo stesso oggetto. Il predicato is/2 consente di rispondere affermativamente alla query "5 is 2+3" e alla query "A is 2+3" con "A=5, yes" ma non si comporta correttamente nel caso in cui si ponga la query "5 is A+3" rispondendo "A=2."

Lo schema CLP(X) (dove X è il generico dominio di computazione) permette la definizione di oggetti semanticamente appartenenti ad X, di operazioni primitive e di relazioni (vincoli). Su questi si può:

- Estrarre dalle relazioni $Y=2+3$, $5=A+3$ le informazioni corrette sull'istanziazione della variabile libera.
- Trattare relazioni del tipo $X+3=Y-2$.
- Mantenere questo vincolo in forma implicita fino a quando una istanziazione di una delle due variabili libere fornisce informazioni sull'altra.

Tra i principali domini di computazione per i quali è stato costruito un constraint solver troviamo i **reali** e lo schema CLP(R), i **razionali** con CLP(Q), gli **interi** CLP(Z), i **booleani** e i **domini finiti** (presente in SWIProlog). I **domini sono usati come strumento realizzativo delle Tecniche di Consistenza**.

Le **Tecniche di Consistenza** possono essere realizzate con la PL costruendo programmi strutturati in modo tale da garantire il pruning a priori dell'albero decisionale. Gli insiemi di valori ammissibili per ciascuna variabile possono essere trattati utilizzando le liste e alcune primitive per agire su di esse. Tuttavia le operazioni sulle liste sono molto pesanti computazionalmente.

Un algoritmo che realizza il **Forward Checking** con gli strumenti offerti dalla PL risulta, in media, è meno efficiente di uno che utilizza lo **Standard Backtracking** per problemi di dimensione ridotta ($n < 12$).

Lo schema CLP si può usare per estendere la PL con meccanismi specifici per il trattamento delle Tecniche di Consistenza e per introdurre il **concetto di dominio associato ad una variabile che ne definisce un range cioè un insieme di valori ammissibili**. La realizzazione dei domini deve essere tale da permettere un loro utilizzo efficiente quindi deve essere svincolata dalle liste e dal loro trattamento.

DOMINIO NELLA CLP

Supponiamo di avere a disposizione una **primitiva domain(X,D)** che associa alla variabile X un insieme di valori possibili contenuto in D, e una **primitiva indomain(X)** che istanzia X, in modo backtrackabile, con un elemento del suo dominio. Un vincolo su una variabile con dominio agisce sul dominio stesso riducendolo.

Es. :- domain(X,[1,2,3,4,5]), X>3. ha come effetto la riduzione del dominio di X a [4,5] mentre il vincolo X>7 conduce al fallimento non essendoci, nel dominio X, alcun valore che soddisfa il vincolo imposto.

Arricchendo la PL in questo modo è possibile esprimere relazioni tra le variabili con dominio

:- domain(X,[1,2,3,4,5]), domain(Y,[3,4,5,6]), X=Y.

L'unificazione delle variabili con dominio deve essere opportunamente gestita e la query precedente effettua la riduzione dei domini di X e di Y ai soli valori [3,4,5], **intersezione dei due domini**.

11 - PLANNING

La **pianificazione automatica (planning)** rappresenta un'importante **attività di problem solving** che consiste nel sintetizzare una sequenza di azioni che eseguite da un agente a partire da uno stato “iniziale” del mondo provocano il raggiungimento di uno stato “desiderato”.

Dati:

- Uno stato iniziale
- Un insieme di azioni eseguibili
- Un obiettivo da raggiungere (goal)

un problema di pianificazione consiste nel **determinare un piano**, ossia un insieme (parzialmente o totalmente) ordinato di azioni necessarie per raggiungere il goal. Pianificare è:

- Una applicazione di per sé
- Un’attività comune a molte applicazioni quali
- Diagnosi: pianificazione di test o azioni per riparare (riconfigurare) un sistema
- Scheduling
- Robotica

Es. Pianificare è una attività che tutti gli esseri umani svolgono nel loro quotidiano. Supponiamo di avere come obiettivo (goal) di seguire una lezione di Intelligenza Artificiale e di essere attualmente a casa e di possedere una macchina (stato iniziale). Al fine di raggiungere lo scopo prefisso dobbiamo fare una serie di azioni in una certa sequenza:

1. prendere materiale necessario per gli appunti,
2. prendere le chiavi della macchina,
3. uscire di casa,
4. prendere la macchina,
5. raggiungere la facoltà,
6. entrare in aula e così via.

Pianificare ci permette di fare le azioni giuste nella sequenza giusta: non possiamo pensare di invertire l’ordine delle azioni 2 e 3. Per alcune di queste azioni non è necessario pianificare l’ordine (**il piano può contenere un ordinamento parziale**). Invertendo l’ordine delle azioni 1 e 2 si ottiene un piano corretto analogo al precedente.

Ci possono essere piani alternativi per raggiungere lo stesso obiettivo (posso pensare di andare in facoltà a piedi o in autobus).

11.1 - RAPPRESENTAZIONI

Un **pianificatore automatico** è un agente intelligente che opera in un certo dominio e che **date**:

- Una rappresentazione dello stato iniziale
- Una rappresentazione del goal
- Una descrizione formale delle azioni eseguibili

sintetizza dinamicamente il piano di azioni necessario per raggiungere il goal a partire dallo stato iniziale.

11.1.1 – RAPPRESENTAZIONE DELLO STATO

Occorre fornire al pianificatore un **modello del sistema** su cui opera. In genere lo stato è rappresentato in forma dichiarativa con una congiunzione di **formule atomiche** che esprime la situazione di partenza.

Es: on(book,table) ^ name(book,xyz) ^ atHome(table)

Lo stato di un sistema spesso può essere osservato solo in modo parziale per una serie di motivi:

- Alcuni aspetti non sono osservabili
- Il dominio è troppo vasto per essere rappresentato nella sua interezza (**limitate capacità computazionali** per la rappresentazione)
- Le osservazioni sono soggette a rumore e quindi si hanno delle osservazioni parziali o imperfette
- Il dominio è troppo dinamico per consentire un aggiornamento continuo della rappresentazione

11.1.2 – RAPPRESENTAZIONE DEL GOAL

Per rappresentare il goal si utilizza lo stesso linguaggio formale con cui si esprime lo stato iniziale. In questo caso la congiunzione rappresenta una descrizione parziale dello stato finale che si vuole raggiungere: descrive solo le condizioni che devono essere verificate affinché il goal sia soddisfatto.

11.1.3 – RAPPRESENTAZIONE DELLE AZIONI

È necessario fornire al pianificatore una **descrizione formale delle azioni eseguibili detta Teoria del Dominio**. Ciascuna azione è identificata da un nome e modellata in forma dichiarativa per mezzo di pre-condizioni e post-condizioni. Le pre-condizioni rappresentano le condizioni che devono essere verificate affinché l'azione possa essere eseguita; le post-condizioni rappresentano gli effetti dell'azione stessa sul mondo. Spesso la Teoria del Dominio è costituita da operatori con variabili che definiscono classi di azioni. A diverse istanziazioni delle variabili corrispondono diverse azioni.

Es. mondo dei blocchi. Problema: spostare blocchi su un tavolo con un braccio. Azioni:

(1)		STACK(X,Y) SE: holding(X) and clear(Y)	(2)		UNSTACK(X,Y) SE: handempty and clear(X) and on(X,Y)
(2)		ALLORA: handempty and clear(X) and on(X,Y);	(1)		ALLORA: holding(X) and clear(Y);
(1')		PICKUP(X) SE: ontable(X) and clear(X) and handempty ALLORA: holding(X);	(2')		PUTDOWN(X) SE: holding(X) ALLORA: ontable(X) and clear(X) and handempty
(2')			(1')		

11.2 - PIANIFICAZIONE

Il processo per il calcolo dei passi di una procedura di soluzione di un problema di pianificazione può essere:

- **Non decomponibile** perché ci può essere interazione fra i sotto problemi
- **Reversibile** se le scelte fatte durante la generazione del piano sono revocabili (**backtracking**).

Un pianificatore è **completo** quando riesce sempre a trovare la soluzione, se esiste. Un pianificatore è **corretto** quando la soluzione trovata porta dallo stato iniziale al goal finale in modo consistente con eventuali vincoli.

Esistono due **macro-categorie di pianificazione**: quella **classica** e quella **reattiva**.

11.2.2 – PIANIFICAZIONE CLASSICA

La pianificazione classica è un tipo di pianificazione off-line che produce l'intero piano prima di eseguirlo lavorando su una rappresentazione istantanea (snapshot) dello stato corrente. È basata su **assunzioni forti**:

- **Tempo atomico** di esecuzione delle azioni
- **Determinismo degli effetti**
- **Stato iniziale completamente noto a priori**
- **Esecuzione del piano unica causa di cambiamento del mondo**

Esistono diverse **Tecniche di Pianificazione Classica**:

- **Planning Deduttivo**
 - Situation Calculus
- **Planning mediante ricerca**
 - Ricerca nello spazio degli stati: Planning Lineare
 - STRIPS

Quello che cambia è lo spazio di ricerca, definito da che cosa sono gli stati e gli operatori:

- **Pianificazione deduttiva come theorem proving:**
stati come insiemi di formule e operatori come regole di inferenza
- **Pianificazione nello spazio degli stati:**
stati come descrizioni di situazioni e operatori come modifiche dello stato
- **Pianificazione nello spazio dei piani:**
stati come piani parziali e operatori di raffinamento e completamento di piani

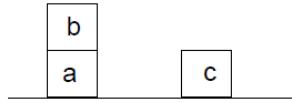
11.2.3 - PLANNING DEDUTTIVO

La tecnica di pianificazione deduttiva utilizza la logica per rappresentare stati, goal e azioni e genera il piano come dimostrazione di un teorema Formulazioni di Green e Kowalsky.

Es. ho un tavolo con tre blocchi. Facciamo una modellazione molto semplice in cui non consideriamo la posizione del tavolo nello spazio o l'area del tavolo visto dall'alto. Consideriamo anche che i blocchi siano tutti uguali: se avessi un cono, sappiamo di non poterlo mettere sotto ad un blocco. Quindi in questo modello molto semplice ho questi predicati:

on(A,B) ontable(A) Obiettivo: mettere b su c.

Devo scrivere che A è sopra al tavolo, che B è sopra ad A e che C è sul tavolo. Come faccio a superare il problema grosso della logica che è l'assioma della monotonicità (se una cosa è vera, è vera e basta)? Se nella mia logica scrivo che b è sopra ad a, come mappo l'operatore che prende b e lo sposta e lo mette su c?



Se b è sopra ad a, ma sono nella fase di planning, b è sopra ad a nello stato iniziale. Sono stato poco preciso: non è una verità scolpita nella roccia, questo è vero solo nello stato iniziale. Aggiungiamo uno stato con tre argomenti, ad esempio, che dice:

on(A, B, STATO)

Allora posso dire che questa cosa è vera nello stato S0. Calcolo delle situazioni.

SITUATION CALCULUS

Situation Calculus è una **tecnica di pianificazione classica** che fa parte del **planning deduttivo**. È una **formalizzazione del linguaggio (basato sulla logica dei predicati del primo ordine)** in grado di rappresentare **stati e azioni in funzione del tempo**. Situation significa "fotografia" del mondo e delle **proprietà (fluent)** che valgono in un determinato istante/stato s. (Es: mondo dei blocchi con $\text{on}(b,a,s)$ e $\text{ontable}(c,s)$). Le azioni definiscono quali fluent saranno veri come risultato di un'azione.

Es. $\text{on}(X,Y,S)$ and $\text{clear}(X,S) \rightarrow (\text{ontable}(X,\text{do}(\text{putOnTable}(X),S)))$ and $(\text{clear}(Y,\text{do}(\text{putOnTable}(X),S)))$

Pre-condizioni: ci sono un blocco x e un blocco y e su x non c'è nulla. Allora posso transitare in un nuovo stato in cui: x è ontable (sto prendendo un blocco e lo appoggio sul tavolo) e in quel nuovo stato su y non ci sarà più nulla. **Ogni azione ci porta in un nuovo stato.** Qual è la nuova situazione? Non sarà più S ma sarà un nuovo stato che io chiamo con un funtore. In base allo stato quindi, ci darà tutta la lista di azioni legate a quello stato.

Per costruire un piano ci servono la deduzione e la dimostrazione di un goal. Es. $\text{:}- \text{ontable}(b,S)$. significa: esiste uno stato S in cui è vero $\text{ontable}(b)$. \rightarrow YES per $S=\text{putOntable}(b,s)$.

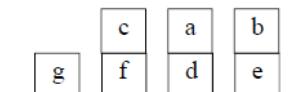
Vantaggi: elevata espressività, permette di descrivere problemi complessi.

Problema: frame problem \rightarrow Occorre specificare esplicitamente tutti i fluent che cambiano dopo una transizione di stato e anche quelli che NON cambiano (assiomi di sfondo: "Frame axioms"). Al crescere della complessità del dominio il numero di tali assiomi cresce enormemente. Il problema della rappresentazione della conoscenza diventa intrattabile.

GREEN

Green usa il Situation Calculus per costruire un pianificatore basato sul metodo di risoluzione. Si cerca la prova di una formula contenente una variabile di stato che alla fine della dimostrazione sarà istanziata al piano di azioni che permette di raggiungere l'obiettivo. I seguenti assiomi descrivono tutte le relazioni vere nello stato iniziale s0:

A.1 $\text{on}(a,d,s0)$. A.2 $\text{on}(b,e,s0)$. A.3 $\text{on}(c,f,s0)$. A.4 $\text{clear}(a,s0)$. A.5
 $\text{clear}(b,s0)$. A.6 $\text{clear}(c,s0)$. A.7 $\text{clear}(g,s0)$. A.8 $\text{diff}(a,b)$. A.9 $\text{diff}(a,c)$.
A.10 $\text{diff}(a,d)$



Le azioni si esprimono con assiomi nella forma a clausole.

L'azione $\text{move}(X,Y,Z)$ è l'azione che trasla il nostro blocco (b è sopra ad a e lo metto sopra a c).

$\text{clear}(X,S)$ and $\text{clear}(Z,S)$ and $\text{on}(X,Y,S)$ and $\text{diff}(X,Z)$.

$\rightarrow \text{clear}(Y,\text{do}(\text{move}(X,Y,Z),S))$, $\text{on}(X,Z,\text{do}(\text{move}(X,Y,Z),S))$. \rightarrow Questa sposta un blocco X da Y a Z, partendo dallo stato S e arriva allo stato $\text{do}(\text{move}(X,Y,Z),S)$. Si esprime con i seguenti assiomi (effect axioms):

A.11 $\sim \text{clear}(X,S)$ or $\sim \text{clear}(Z,S)$ or $\sim \text{on}(X,Y,S)$ or $\sim \text{diff}(X,Z)$ or $\text{clear}(Y,\text{do}(\text{move}(X,Y,Z),S))$.
A.12 $\sim \text{clear}(X,S)$ or $\sim \text{clear}(Z,S)$ or $\sim \text{on}(X,Y,S)$ or $\sim \text{diff}(X,Z)$ or $\text{on}(X,Z,\text{do}(\text{move}(X,Y,Z),S))$.

Dato un goal vediamo un esempio di come si riesce a trovare una soluzione usando il metodo di risoluzione:

GOAL $\text{:}- \text{on}(a,b,S1) \rightarrow \sim \text{on}(a,b,S1)$

(A.12) {X/a,Z/b,S1/do(move(a,Y,b),S)}

$\sim \text{clear}(a,S)$ or $\sim \text{clear}(b,S)$ or $\sim \text{on}(a,Y,S)$ or $\sim \text{diff}(a,b)$

(A.4) (A.5) (A.1) (A.8)

{S/s0}, {S/s0}, {S/s0, Y/d} true

$\sim \text{on}(a,b,S1)$ porta a una contraddizione quindi $\text{on}(a,b,S1)$ risulta dimostrato con la sostituzione S1/do(move(a,d,b),s0).

Supponiamo di voler risolvere un problema un po' più complesso Goal: $\text{on}(a,b,S)$, $\text{on}(b,g,S)$.

Soluzione: S/do(move(a,d,b),do(move(b,e,g),s0)). Per poterlo risolvere occorre una descrizione completa dello stato risultante dall'esecuzione di ciascuna azione.

Per descrivere un'azione oltre agli effect axioms occorre specificare tutti i fluent che NON sono invalidati dall'azione stessa (frame axioms). Nel nostro esempio occorrono i seguenti assiomi:

$\text{on}(U,V,S) \text{ and } \text{diff}(U,X) \rightarrow \text{on}(U,V,\text{do}(\text{move}(X,Y,Z),S))$

$\text{clear}(U,S) \text{ and } \text{diff}(U,Z) \rightarrow \text{clear}(U,\text{do}(\text{move}(X,Y,Z),S))$

Occorre esplicitare un frame axioms per ogni relazione NON modificata dall'azione → Se il problema è complicato la complessità diventa inaccettabile.

11.2.4 – PLANNING MEDIANTE RICERCA

La pianificazione classica non deduttiva **utilizza linguaggi specializzati** per rappresentare stati, goal e azioni e **gestisce la generazione del piano come un problema di ricerca (search)**.

La ricerca può essere effettuata:

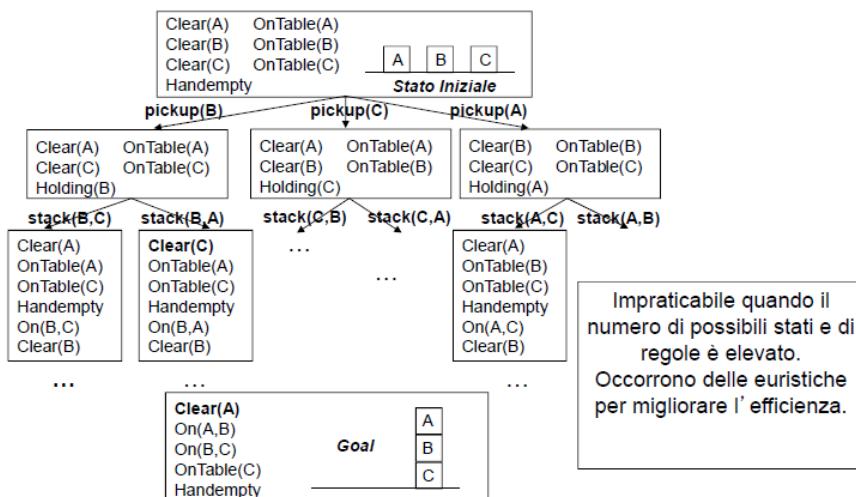
- **Nello spazio degli stati o situazioni:** nell'albero di ricerca ogni nodo rappresenta uno stato e ogni arco un'azione
- **Nello spazio dei piani:** nell'albero di ricerca ogni nodo rappresenta un piano parziale e ogni arco un'operazione di raffinamento del piano

RICERCA NELLO SPAZIO DEGLI STATI: PLANNING LINEARE

Un pianificatore lineare riformula il problema di pianificazione come problema di ricerca nello spazio degli stati e utilizza le strategie di ricerca classiche. L'algoritmo di ricerca può essere:

- **Forward:** se la ricerca avviene in modo progressivo partendo dallo stato iniziale fino al raggiungimento di uno stato che soddisfa il goal.
- **Backward:** quando la ricerca è attuata in modo regressivo a partire dal goal fino a ridurre il goal in sotto-goal soddisfatti dallo stato iniziale.

Es. Ricerca Forward: il linguaggio di questi sistemi è un linguaggio logico, ma è separato dalla logica. **Questi sistemi formalizzano un linguaggio pseudologico.**



STRIPS (Stanford Research Institute Problem Solver)

È l'antenato degli attuali sistemi di pianificazione ed è un **linguaggio per la rappresentazione di azioni**. La sintassi è molto più semplice del **Situation Calculus** (meno espressività, più efficienza). È un **algoritmo per la costruzione di piani**. Sostanzialmente ci dà già un linguaggio che ha un motore di inferenza che non è forward ma backward con un'attenzione particolare all'ordine dei goal.

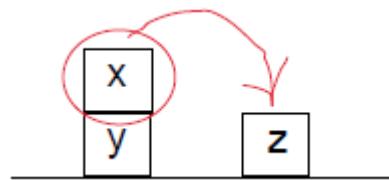
LINGUAGGIO STRIPS

Le caratteristiche del linguaggio sono le seguenti:

- La **rappresentazione dello stato** è fatta da un insieme di fluent che valgono nello stato. (Es: on(b,a), clear(b), clear(c), ontable(c))
- La **rappresentazione del goal** è un insieme di fluent (simile allo stato) e può avere variabili (Es: on(X,a))
- La **rappresentazione delle azioni** è formata da tre liste
 - **Precondizioni**: fluent che devono essere veri per applicare l'azione
 - **Delete List**: fluent che diventano falsi come risultato dell'azione
 - **Add List**: fluent che diventano veri come risultato dell'azione

Es. Move(X, Y, Z), Precondizioni: on(X,Y), clear(X), clear(Z)

Delete List: clear(Z), on(X,Y), Add list: clear(Y), on(X,Z)



A volte Add e Delete list sono rappresentate come EFFECT list con atomi positivi e negativi.

Es. Move(X, Y, Z), Precondizioni: on(X,Y), clear(X), clear(Z)

Effect List: $\neg\text{clear}(Z)$, $\neg\text{on}(X,Y)$, $\text{clear}(Y)$, $\text{on}(X,Z)$

Il **frame problem** risolto con la Strips Assumption: tutto ciò che non è specificato nella Add e Delete list resta immutato.

Es. del mondo a blocchi

AZIONI in STRIPS:

pickup(X), PRECOND: ontable(X), clear(X), handempty, DELETE: ontable(X), clear(X), handempty, ADD: holding(X)

putdown(X), PRECOND: holding(X), DELETE: holding(X), ADD: ontable(X), clear(X), handempty

stack(X,Y), PRECOND: holding(X), clear(Y), DELETE: holding(X), clear(Y), ADD: handempty, on(X,Y), clear(X)

unstack(X,Y), PRECOND: handempty, on(X,Y), clear(X), DELETE: handempty, on(X,Y), clear(X), ADD: holding(X), clear(Y)

ALGORITMO STRIPS

Nell'algoritmo Strips il **planner lineare** è basato su ricerca **backward** e si assume che lo stato iniziale sia completamente noto (**Closed World Assumption**).

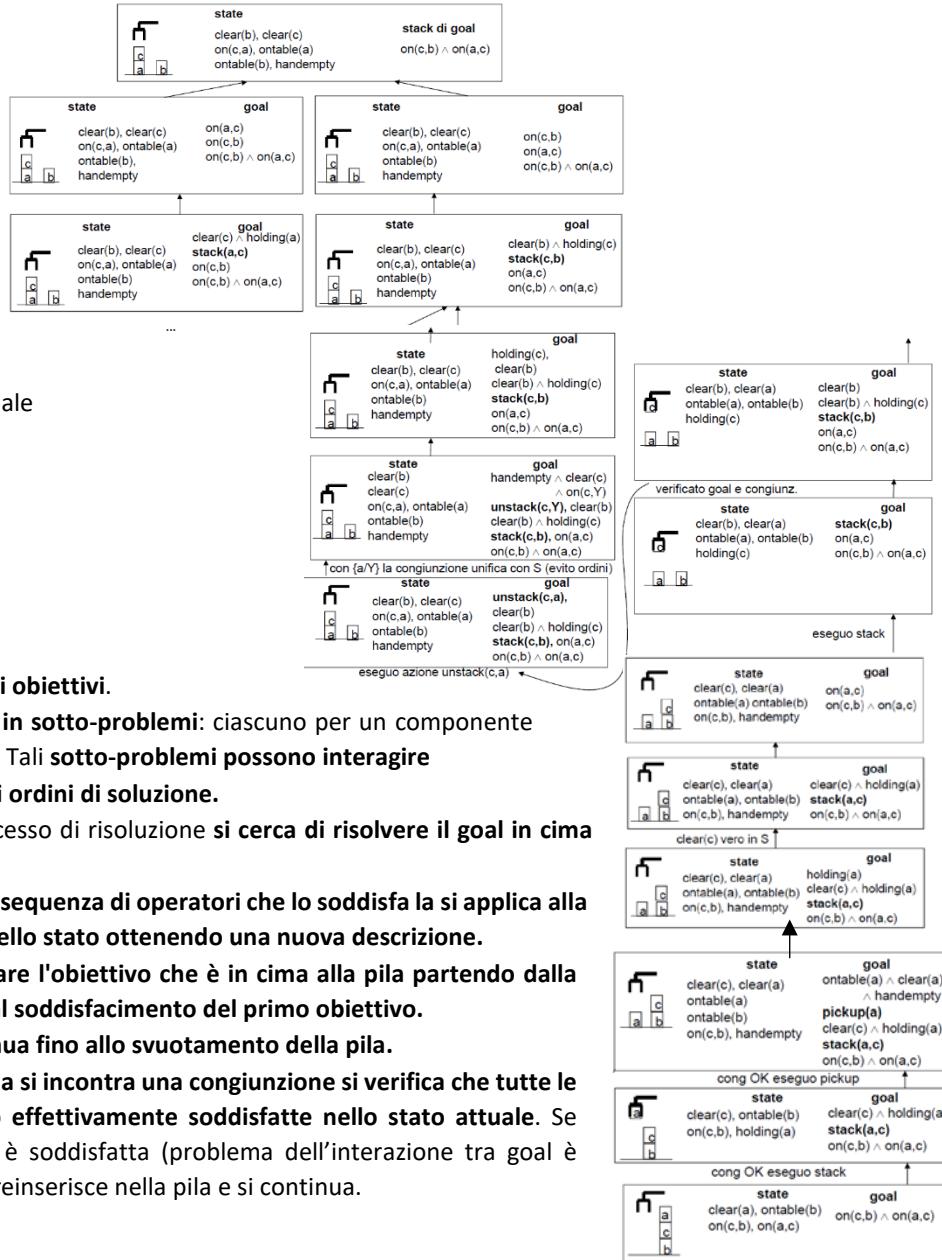
L'algoritmo utilizza due strutture dati:

- Stack di goal
- Descrizione S dello stato corrente

L'Algoritmo:

```
- Inizializza stack con la congiunzione di goal finali
- while stack non è vuoto do
    if top(stack) = A and  $A \theta \sqsubseteq S$  (si noti che A puo essere un and di goals o un atomo)
        then pop(a) ed esegui sost θ sullo stack
        else if top(stack) = a
            then
                - seleziona regola R con a ∈ Addlist(R),
                - pop(a), push(R), push(Precond(R));
                else if top(stack) = a1 ∧ a2 ∧ ... ∧ an
                    (*) then push(a1),..., push(an)
                    else if top(stack) = R
                        then pop(R) e applica R trasformando S
    (*) si noti che l' ordine con cui i sottogoal vengono inseriti nello stack
        rappresenta un punto di scelta non deterministica. La congiunzione
        rimane sullo stack e verrà riverificata dopo - interacting goals
```

Es.



Ricostruendo da configurazione iniziale END a finale ho una soluzione:

1. unstack(c,a)
2. stack(c,b)
3. pickup(a)
4. stack(a,c)

Considerazioni sull'algoritmo:

- Il **goal è la prima pila di obiettivi**.
- **Suddivide il problema in sotto-problemi**: ciascuno per un componente dell'obiettivo originale. Tali **sotto-problemi possono interagire**
- Abbiamo tanti possibili ordini di soluzione.
- Ad ogni passo del processo di risoluzione si cerca di risolvere il **goal in cima alla pila**.
- Quando si ottiene una sequenza di operatori che lo soddisfa la si applica alla descrizione corrente dello stato ottenendo una nuova descrizione.
- Si cerca poi di soddisfare l'obiettivo che è in cima alla pila partendo dalla situazione prodotta dal soddisfacimento del primo obiettivo.
- Il procedimento continua fino allo svuotamento della pila.
- Quando in cima alla pila si incontra una congiunzione si verifica che tutte le sue componenti siano effettivamente soddisfatte nello stato attuale. Se una componente non è soddisfatta (problema dell'interazione tra goal è spiegato più avanti) si reinserisce nella pila e si continua.

Problemi

1. **Grafo di ricerca molto vasto**. Nell'esempio abbiamo visto un cammino ma in realtà ci sono varie alternative.
 - a. Scelte non deterministiche ordinamento dei goal
 - b. Più operatori applicabili per ridurre un goal

Soluzione: Strategie euristiche

- **Strategie di ricerca**
- **Strategie per scegliere quale goal ridurre e quale operatore**
 - **MEANS-ENDS ANALYSIS** per:
 - Cercare la differenza più significativa tra stato e goal
 - Ridurre quella differenza per prima
- 2. **Problema dell'interazione tra goal**. Quando due (o più) goal inter-agiscono ci possono essere problemi di interazione tra le due soluzioni.

Es. Goal G1, G2 \rightarrow pianifico azioni per G2, poi per risolvere G1 potrei dover smontare tutto, compreso lo stato che avevo prodotto con G2 risolto.

Soluzione completa: provare tutti gli ordinamenti dei goal e dei loro sottogoal.

Soluzione pratica (Strips):

- Provare a risolverli indipendentemente
- Verificare che la soluzione funzioni
- Se non funziona, provare gli ordinamenti possibili uno alla volta

PIANIFICAZIONE NON LINEARE

In generale, una soluzione può essere quella di utilizzare la Pianificazione Non Lineare. I pianificatori non lineari sono algoritmi di ricerca che gestiscono la generazione di un piano come un problema di ricerca nello spazio dei piani e non più degli stati. L'algoritmo non genera più il piano come una successione lineare (completamente ordinata) di azioni per raggiungere i vari obiettivi.

11.2.5 – PIANIFICAZIONE IN PRATICA

Esistono molte applicazioni in domini complessi:

- Pianificatori per Internet (ricerca di informazioni, sintesi di comandi Unix)
- Gestione di strumentazione spaziale
- Robotica
- Graphplan sviluppato dalla Carnegie Mellon University
- Piani di produzione industriale
- Logistica

Gli algoritmi visti presentano problemi di efficienza in caso di domini con molti operatori e le tecniche per rendere più efficiente il processo di pianificazione prevedono il **Planning Gerarchico**.

11.3 - ESECUZIONE

Il processo di applicazione della procedura di soluzione può essere:

- **Irreversibile** se l'esecuzione delle azioni determina spesso un cambiamento di stato non reversibile,
- **Non deterministica** se il piano può avere un effetto diverso quando applicato al mondo reale che è spesso impredicibile. In questo caso è possibile rifare il piano solo parzialmente, oppure invalidarlo tutto a seconda del problema.

I pianificatori visti finora permettono di costruire piani che vengono poi eseguiti da un agente "esecutore". Possibili problemi in esecuzione:

- **Esecuzione di un'azione in condizioni diverse da quelle previste dalle sue precondizioni**
 - Conoscenza incompleta o non corretta
 - Condizioni inaspettate
 - Trasformazioni del mondo per cause esterne al piano
- **Effetti delle azioni diversi da quelli previsti**
 - Errori dell'esecutore
 - Effetti non deterministicci, impredicibili

Occorre che l'esecutore sia in grado di percepire i cambiamenti e agire di conseguenza. Alcuni pianificatori fanno l'**ipotesi di mondo aperto (Open World Assumption)** ossia considerano l'informazione non presente nella rappresentazione dello stato come non nota e non falsa diversamente dai pianificatori che lavorano nell'ipotesi di mondo chiuso (Closed World Assumption).

Alcune informazioni non note possono essere cercate tramite azioni di "raccolta di informazioni" (azioni di sensing) aggiunte al piano. Le azioni di sensing sono modellate come le azioni causali.

Le pre-condizioni rappresentano le condizioni che devono essere vere affinché una certa osservazione possa essere effettuata, le post-condizioni rappresentano il risultato dell'osservazione.

Due possibili approcci:

- Planning Condizionale
- Integrazione fra Pianificazione ed Esecuzione

11.3.1 – PIANIFICAZIONE REATTIVA

La pianificazione reattiva è un metodo di pianificazione on-line che:

- Considera l'ambiente non deterministico e dinamico
- È capace di osservare il mondo sia in fase di pianificazione, per l'acquisizione di informazione non nota, sia in fase di esecuzione
- Spesso alterna il processo di pianificazione a quello di esecuzione reagendo ai cambiamenti di stato
- Monitora l'esecuzione delle azioni e ne verifica gli effetti

Planning Reattivo (Brooks - 1986). Discendono dai sistemi reattivi "puri" che evitano del tutto la pianificazione ed utilizzano semplicemente la situazione osservabile come uno stimolo per reagire.

11.3.2 - SISTEMI REATTIVI PURI

Hanno accesso ad una base di conoscenza che descrive quali azioni devono essere eseguite ed in quali circostanze.

Scelgono le azioni una alla volta, senza anticipare e selezionare un'intera sequenza di azioni prima di cominciare.

Es. Termostato utilizza le semplici regole:

- 1) Se la temperatura T della stanza è K gradi sopra la soglia T0, accendi il condizionatore;
- 2) Se la temperatura della stanza è K gradi sotto T0, spegni il condizionatore.

Vantaggi:

- Sono capaci di interagire con il sistema reale. Essi operano in modo robusto in domini per i quali è difficile fornire modelli completi ed accurati.
- Non usano modelli, ma solo l'immediata percezione del mondo e per questo sono anche estremamente veloci nella risposta.

Svantaggio: Il loro comportamento in domini che richiedono di ragionare e deliberare in modo significativo è deludente (es. scacchi) in quanto non sono in grado di generare autonomamente piani.

11.3.3 - PIANIFICATORI IBRIDI

I moderni pianificatori reattivi detti ibridi integrano approccio generativo e approccio reattivo al fine di sfruttare le capacità computazionali del primo e la capacità di interagire con il sistema del secondo affrontando così il problema dell'esecuzione. Un pianificatore ibrido:

- Genera un piano per raggiungere il goal
- Verifica le precondizioni dell'azione che sta per eseguire e gli effetti dell'azione appena eseguita
- Smonta gli effetti di un'azione (importanza della reversibilità delle azioni) e ripianifica in caso di fallimenti
- Corregge i piani se avvengono azioni esterne impreviste