Risposte a tutte le possibili domande fatte all'orale (almeno crediamo) da parte di
Lieto-Micalizio-Pozzato.

A cura di : Mario Scapellato, Gabriele Naretto, Abbas Abdirashid

A.A. 21/22.

# DOMANDE LIETO

1) Come si può distinguere un sistema cognitivo e intelligenza ispirato?
    - 
2) Nouvelle AI vs cognitivismo
    - Sono le principali prospettive storiche per quanto riguarda la costruzione di sistemi intelligenti. L'Idea alla base del cognitivismo e' che l'intelligenza, sia negli esseri viventi che negli esseri umani, e' la capacita' di andare a elaborare dei simboli delle strutture simboliche. Si ha quindi:
        - Focus su forme di ragionamento, pianificazione, linguaggio ecc..
        - Le rappresentazioni sono simboliche
        - Si cerca di voler andare a interagire diverse funzionalità' cognitive (risolvere più task).
        - Ispirazione cognitiva

      Per quanto riguarda la Novelle AI, esso ha:
        - Focus sulla percezione
        - Rappresentazioni Distribuite
        - Prospettive di singolo sistema → Un singolo problema risolto in modo ottimale (singolo task)
        - Ispirazione biologica
3) Che cosa è poter esplicativo?
    - Una caratteristica del sistema intelligente è che sia esplicativo, ovvero vogliamo utilizzare il sistema ottenuto per capire delle cose nuove rispetto alle teorie iniziali che abbiamo implementato. Vogliamo fare in modo di riuscire a

generare nuova conoscenza rispetto al sistema naturale che abbiamo preso come fonte d'ispirazione.
- Un sistema cognitivo deve idealmente avere la capacità di compiere inferenza inversa/un potere esplicativo. In questo senso intendiamo che quando andiamo a creare un sistema cognitivo basandosi su un sistema naturale vogliamo poi essere in grado di scoprire diverse informazioni sul funzionamento del sistema naturale che abbiamo preso di esempio.

## 4) Minimal cognitive grid e i suoi limiti (LA CHIEDE SEMPRE)

- Permette di collocare i diversi programmi che vengono sviluppati dai sistemi cognitivi all'interno di una griglia. Tale griglia viene suddivisa in 3 dimensioni:
  - **Ratio**: considera sia quanti vincoli abbiamo, ma anche il rapporto tra gli elementi funzionali ed elementi strutturali
  - **Generality:** indica quante cose riesco a fare con un certo tipo di architettura, ovvero la capacità di un sistema di svolgere più funzioni e non essere orientati a un solo task.
  - **Performance Match:** significa andare a vedere sia i successi, ma anche come sbagliano i sistemi biologici presi come fonte d'ispirazione
  - i suoi limiti potrebbero essere il fatto che questi tre elementi siano un indicatore per classificare un modello costruito su questi paradigmi. ci possono essere casi in cui per esempi il match umano-modello varia molto in base all'umano a cui facciamo riferimento, oppure al tipo di task che il modello compie e cosa intendiamo come task.

## 5) Cognitive band

- Il comportamento di un agente intelligente può' essere descritto da attività' che possono essere suddivise in diversi tipi di **band biologica cognitiva razionale e sociale**. Per **cognitive band,** intendiamo tutte quelle azioni che intercorrono tra i 100 millisecondi e i 10 secondi; per **band razionali,** intendiamo tutte quelle azioni che intercorrono tra alcuni minuti e ci impiegano ore; per **band sociale,** intendiamo meccanismi di computazione che impiegano da qualche giorno fino a mesi.
- Questo sistema è come funziona il sistema di esecuzione di SOAR.

## 6) Se gli approcci neurali sono più cognitivamente o biologicamente ispirati e perché?

- Falso, perché per rendere cognitivamente o biologicamente valido un programma che noi andiamo a costruire, l'elemento importante che dobbiamo considerare sono i vincoli aggiuntivi che vengono dalla biologia, oppure dal cognitivo che andiamo a sovra-imporre su questi tipi di programmi.
- Gli approcci neurali sono un chiaro esempio di paradigma funzionalista biologicamente ispirato. Poiché noi replichiamo la struttura della neuroni reali (ma con delle piccole variazioni, come il caso della back propagation che nella realtà non esiste) cercando di ottenere un sistema che lavora bene su un singolo task e che abbia un output molto ottimizzato.

## 7) 5 passi della resource rationality

- Sono i seguenti:
    - Si parte dal livello computazionale e si analizza un problema in termini di goal, quali sono gli input e output
    - si scende e si inizia a vedere quali sono le classi di algoritmi che posso utilizzare per risolvere un problema
    - Una volta elencati tutti gli algoritmi, cerco di vedere quali sono gli algoritmi che hanno un trade off migliore tra le risorse computazionali che devono essere utilizzate e la cura che viene gestita per andare a modellare quel tipo di problema
    - Cerco di vedere se ci sono delle discrepanze tra le assunzioni fatte nel mio modello e quelli che sono i risultati sperimentali che si trovano in altre discipline
    - Una volta fatto il confronto, cerco di organizzare il mio modello in modo che la performance match sia sempre piu' significativa

## 8) Architettura di brooks

- L'idea è quella di sfruttare il concetto di usare il sistema deduttivo, ruotando la sua architettura (spostandola orizzontalmente) individuando una serie di attività crescenti che un sistema (ad es. robotico) deve essere in grado di risolvere. Questo tipo di organizzazione architetturale permetteva che, passando da un livello più basso a un livello superiore, non si perdesse quello che si era ottenuto ai livelli precedenti. Si dice **Sussunzione** perché man mano che si sale nelle capacità che vengono acquisite, non si perde ciò che si e' acquisito al livello precedente. Ci sono tre elementi fondamentali di questa proposta:
    - Non c'è nessuna rappresentazione
    - L'agente deve creare di volta in volta la rappresentazione fisica del mondo
    - Si possono usare automi a stati finiti per collegare i livelli, Si usano i modelli a stati finiti.

## 9) Approccio Enactive

- L'idea e' che agente e ambiente si determinano insieme e viene condiviso un ambiente enactive in cui ciascuno di essi si costruisce una vista diversa dell'ambiente. Per comunicare inoltre, gli agenti devono avere completa autonomia con l'esterno. Abbiamo 5 elementi chiave:
    - Autonomia (nessun controllo esterno)
    - Embodiment (corpo)
    - Emergence (cognizione emerge dell'interazione)
    - Esperienza (interazioni passate modificano agente e ambiente)
    - Sense-Making (capacita' di auto-modifica del comportamento)

## 10) Gerarchia Marr

- Permette di andare ad analizzare il comportamento di un qualsiasi sistema artificiale sia cognitivamente ispirato che non, utilizzando tre livelli:
    - **Teoria Computazionale**: si focalizza sugli aspetti di input/output, ovvero qual e' il goal, l'input che gestisce, l'output che devo produrre e le strategie utilizzate
    - **Rappresentazione e algoritmo**: quali sono le strutture dati gli algoritmi che permettono d'implementare la teoria computazionale del livello di sopra
    - **Implementazione hardware e software**: livello più basso.

## 11) Limiti sistemi cognitivisti

- partiamo del presupposto che si basa su un approccio simbolico, il quale e' una teoria dell'intelligenza che vale per esseri viventi e macchine ed è stata smentita da tempo. Il più grande limite se parliamo di sistemi strutturalisti cognitivamente ispirati è dovuto al paradosso di wiener (il miglior sistema e il sistema biologico stesso).

## 12) Chunking

- Si tratta dell'apprendimento di una regola per risolvere un problema. E' un meccanismo per cui imparo una certa regola e la prossima volta che mi ritrovo in uno stato che ha la stessa situazione che ho precedentemente risolto, non vado a rifare tutto il ciclo di SOAR, ma applico tale regola che e' stata spostata automaticamente nella memoria procedurale. Questa procedura di chunking viene appresa da SOAR tramite **backtracking.** A partire da tutte quelle sequenze di stati che hanno portato al superamento dell'impasse, faccio backtracking, vedo quali sono gli stati e gli operatori utilizzati e li metto in una regola che va a finire all'interno della memoria procedurale.

## 13) Analogie e differenze tra SOAR e ACT-R (chi dei due è più strutturalista?)

- Architettura utilizzata soprattutto per modellazioni cognitive, ovvero fare inferenza inversa, mentre in SOAR cerchiamo di creare degli agenti intelligenti. ACT-R e un architettura ibrida, in cui la conoscenza e' simile a SOAR, ma l'attivazione dei chunk avviene grazie a una rete neurale di tipo simbolico.
  In ACT-R abbiamo al centro una memoria procedurale centralizzata e poi una serie di moduli esterni a cui e' associata una micro-memoria a breve termine, mentre SOAR e' un architettura formata da una long term memory e una short term memory e ha vari moduli connessi con elementi percettivi o motori. I moduli in ACT-R hanno un mapping diretto con la conoscenza nota di come funzionano i meccanismi delle diverse parti del cervello. ACT-R ha una corrispondenza strutturale tra il software e gli esperimenti psicologici ed e' usato sia per i modelli cognitivi che per le ricerche in AI.

## 14) Test di Turing e varianti

- Un interrogatore dialoga senza vedere chi c'e' dall'altro lato se una macchina o un essere umano e il test e' superato se l'interlocutore non riesce a distinguere con chi sta dialogando. Si può' vedere questo test come un metodo per attribuire intelligenza a un sistema che significa attribuire la capacita' di risolvere un singolo task. Tuttavia il **Turing Test** presenta dei problemi:
    - Il risultato del test dipende da chi interroga quindi questo e' un problema perché' diversi interrogatori possono avere interpretazioni diverse.
    - Questo Test non e' adatto per l'intelligenza generale perché' e' interamente basato sul linguaggio, quindi sulla competenza linguistica

  Abbiamo poi il **Total Turing Test**, in cui anziché considerare una agente software in grado di rispondere a delle domande, immaginiamo di avere un agente in grado di risolvere una serie di attività' tale per cui diventa possibile non riuscire più a distinguere se un task e' svolto da un umano o da un agente.
  **Super Simplified Turing Test:** non ci sono barriere tra interlocutore e agente l'interlocutore deve valutare le risposte di quest'ultimo.
  Ma anche questa versione ha dei problemi.
  Altri problemi sono:
    - Quanto deve andare avanti la conversazione ?
    - Chi è l'interrogatore ?
    - Di cosa si parla nella conversazione ?
    - Come decide l'interrogatore ?

## 15) Come capire se un sistema ha potere esplicativo e quale spiegazione funziona

- Per poter spiegare questa domanda possiamo fare riferimento a due paradigmi di progettazione:
    - **Approccio Funzionalista**: un sistema artificiale per avere un potere esplicativo rispetto al sistema naturale preso come fonte d'ispirazione, deve avere una sorta di equivalenza debole tra le procedure implementate e i processi cognitivi noti all'interno del sistema naturale.
    - **Approccio strutturalista**: equivalenza forte tra le procedure con cui ho costruito il mio sistema e i corrispondenti meccanismi cognitivi noti all'interno del sistema naturale. Il focus e' anche sui vincoli che devono essere inseriti nel modello, in modo da poter generare una computazione human like.

## 16) Sistema di Google NLP può essere cosciente?

No. Il sistema di google NLP sfrutta algoritmi avanzatissimi di ricerca nelle risorse linguistiche, per estrarre significato da risorse testuali/vocali. Ricordiamo il paradosso di weiner. Pylyshyn, inoltre disse che "Se non formuliamo nessuna restrizione riguardo al

modello otteniamo il funzionalismo di una macchina di turing, se invece applichiamo tutte le possibili restrizioni otteniamo un essere umano".

## 17) Problemi delle reti neurali

## 18) Differenza tra sistema biologicamente ispirato e sistema cognitivamente ispirato

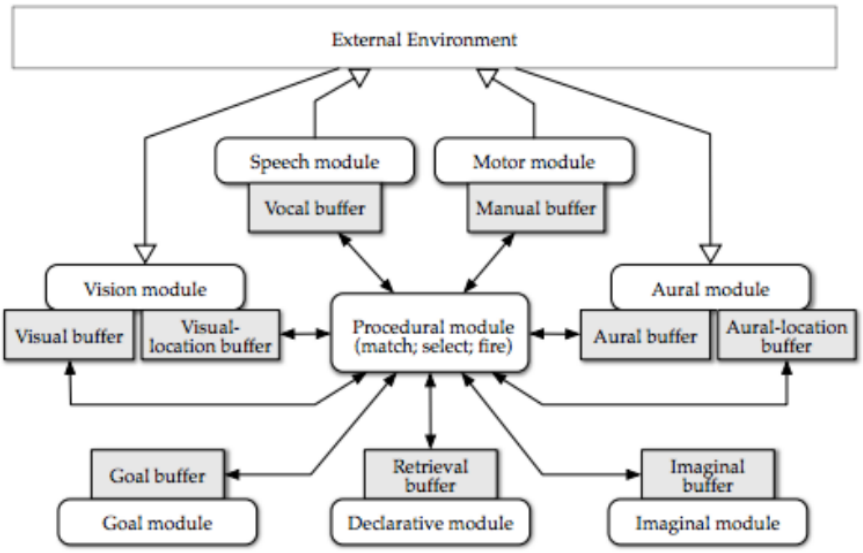
- La differenza principale riguarda il livello di astrazione considerato. I sistemi biologicamente ispirati sono vincolati ad aspetti di tipo neurale, mentre gli aspetti di tipo cognitivo emergano da vincoli strutturali di tipo anatomico, in cui ad esempio sono interessato a vedere se l'informazione che passa da un neurone all'altro della mia rete segue qualche tipo di principio che si ritrova all'interno del cervello.

  Quando mi occupo di sistemi cognitivamente ispirati mi focalizzo su un livello più alto di astrazione, quindi utilizzo dei paradigmi diversi (se ad es. in un caso utilizzo dei sistemi connessionisti, in un altro caso userò' approcci ibridi o simbolici).

## 19) ACT-R (architettura)

Architettura con diversi modelli implementati e sviluppata in lisp.
Struttura diversa di soar dal punto di vista di elaborazione. Al centro abbiamo una struttura centralizzata di regole (procedular module), con tanti moduli esterni.

Struttura diversa rispetto a SOAR; al centro ce una memoria procedurale centralizzata e poi ci sono una serie di moduli esterni a cui e' associata una micro-memoria a breve termine.




Tutto e' guidato da una memoria procedurale; quindi questo modulo e' una parte della memoria a lungo termine ed e' il modulo che controlla gli altri moduli.

Ogni modulo che vediamo nell'immagine ha una propria memoria di lavoro (buffer).
Tutto è controllato da queste regole (accesso hai moduli e operazioni).
Un altra grande differenza con SOAR è che c'è un mapping dei moduli diretto con la struttura del cervello umano(almeno quello che conosciamo oggi).
I diversi moduli con cui funziona l'architettura ha un mapping diretto con la conoscenza nota di come funzionano i meccanismi delle diverse parti del cervello.

Questa architettura viene usata soprattutto per modellazioni cognitive, ovvero fare inferenza inversa, mentre in SOAR cerchiamo di creare agenti intelligenti.

ACT-R è un architettura ibrida, dove la conoscenza si rappresenta simile a soar, ma l'attivazione dei chunk avviene grazie a una rete neurale di tipo simbolico. L'output di questa rete è collegato a un concetto bayesiana, attivo una regola se la probabilità condizionata delle esperienze precedenti me lo consiglia.

Per la parte di attivazione dei chunk rispetto al goal usa più vincoli, come per esempio la frequenza dell'operatore oppure quanto un operatore è importante rispetto a un problema da risolvere → abbiamo cosi un inprinting cognitivo molto forte.

ACT-R permette di avere un attivazione probabilistica delle regole; posso avere un matching imparziale cioe', se questo insieme d'informazioni simboliche puo' essere rilevante per risolvere un goal, allora faccio partire la regola.
Vediamo la regola di attivazione dei chunk.

$$\text{activation} = \text{base activation} + \left( \text{source activation} * \text{associative strength} \right) + \left( \text{mismatch penalty} * \text{similarity value} \right) + \text{noise}$$

Vengono considerati gli elementi dell'attivazione di base; questo aspetto viene associato ad altre due grandezze: la forza associativa → quanto il chunk e' importante per risolvere il task e matching penality → quanto il chunk matcha con la regola.
Soar → mapping cognitivi sui cicli di esecuzione, eseguiamo molto in fretta come gli esseri umani.
SOAR vs ACT-R in ambito cognitivo
Anche ACT-R ha vincoli tempora per pensare,

ACT-R ha modelli molto più meccanicistiche. E nei versanti più strutturalisti.

SOAR e un modello quasi funzionalista.

# DOMANDE POZZATO

## 1) Risoluzione SLD è corretta e completa per tutta la logica o solo per alcuni tipi di clausole? In cosa consiste?

- Generalmente questa strategia non e' completa, ma se vengono utilizzate le clausole di Horn si. La risoluzione SLD ha il seguente funzionamento avendo a disposizione una base di conoscenza K formata da clausole di HORN, applica una strategia linear input, in cui a ogni passo una clausola viene presa dalle K di partenza, mentre l'altra e' sempre clausola risolvente del passo precedente. Il procedimento di risoluzione può' concludersi in tre modi differenti:
  - Successo → se otteniamo una clausola vuota
  - Fallimento Finito → se non possiamo derivare il goal o non riusciamo a ottenere una clausola vuota
  - Fallimento Infinito → se e' sempre possibile derivare nuove clausole risolventi.

## 2) Risoluzione SLD (Albero SLD, Terminazione, Non-Determinismo)

- Possiamo rappresentare le derivazioni partendo dal goal con un albero SLD. La radice e' il goal iniziale e  ogni nodo dell'albero e' una atomo a cui viene applicata la sostituzione dell'MGU. Affinché il procedimento sia deterministico, si procede sempre a partire dal nodo più a sinistra (leftmost) e le clausole vengono considerate nell'ordine in cui sono scritte nel programma. Tuttavia questa strategia non e complete perché se troviamo una soluzione nella parte a destra di un ramo infinito, l'interprete non lo trovo perché entra nel ramo infinito senza mai uscirne.

## 3) Se prendo delle clausole che non sono di horn cosa si mette a rischio tra completezza e correttezza, perché sono importanti le clausole di Horn?

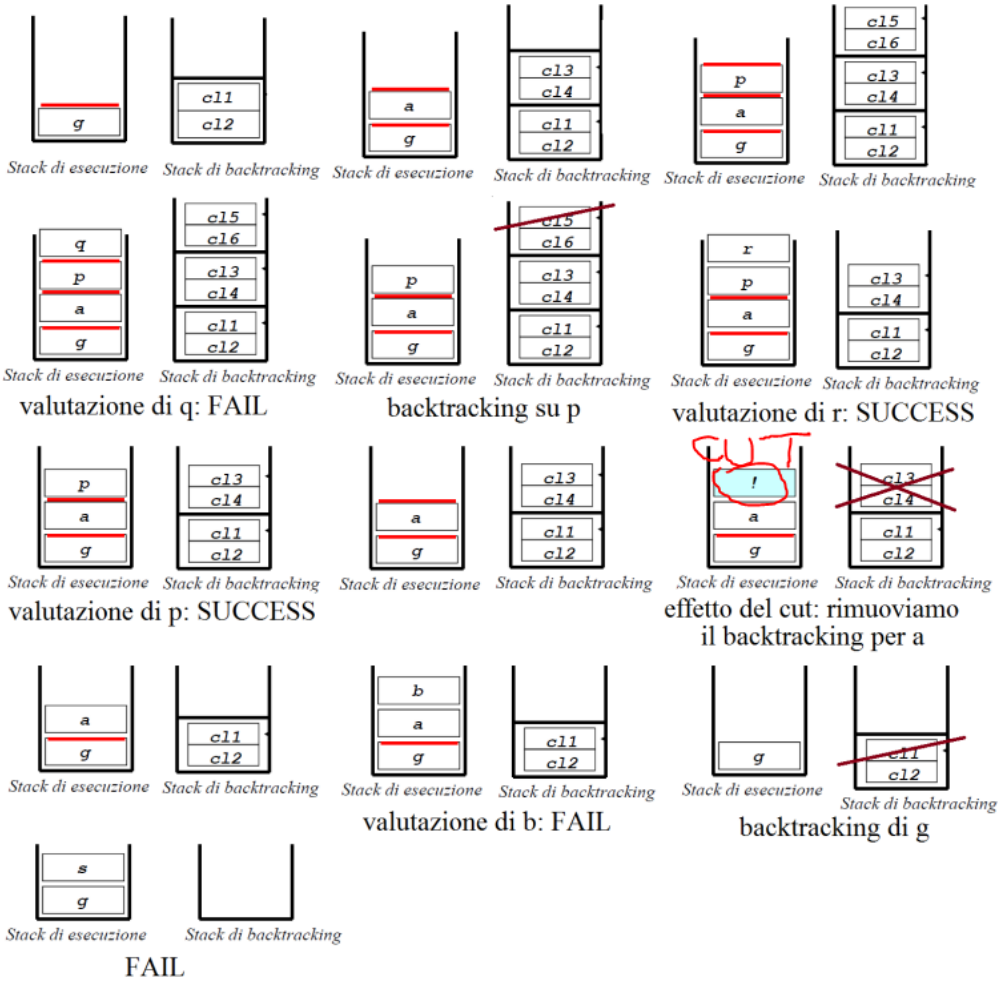
- Metto a rischio entrambe. Il metodo e' corretto e completo solo per clausole di Horn, in cui:
  - Fatti→ essendo clausole unarie, sono di Horn
  - Regole → anche le regole sono clausole di Horn
  - Goal → considerando un goal generico, anche questo sara' un una clausola di Horn.

  Abbiamo quindi la conclusione che tutti gli elementi di un programma in Prolog sono clausole di Horn, dunque il procedimento dimostra essere incompleto e incorretto se questi elementi non venissero presi in considerazione come clausole di Horn.

## 4) Parlare del CUT in prolog, cos'è perché si usa

- Per comprendere al meglio il concetti di CUT, andiamo a studiare il modello a runtime di Prolog:
    - **Stack di esecuzione:** contiene le clausole su cui stiamo operando. In posizione top contiene la clausola su cui stiamo operando in questo momento
    - **Stack di backtracking:** tiene traccia delle possibili regole applicabili per gli elemento contenuti nello stack. In posizione top contiene le regole applicabili per l'elemento posto al top dello stack di esecuzione

Sostanzialmente il CUT e' un predicato extra-logico che serve per controllare e modificare il percorso di risoluzione dell'albero SLD, andando quindi a bloccare punti di backtracking. Inoltre è un predicato sempre vero e aumenta l'efficienza dei programmi Prolog (perché è come se facesse pruning)
Con il CUT andiamo a vincolare le scelte dell'interprete Prolog buttando via punti di backtracking. L'utilizzo del CUT rende definitive le scelte fatte fino a un certo punto andando a rimuovere alcuni elementi dallo stack di backtracking. Nel momento in cui in Prolog si incontra il simbolo di cut !, vengono eliminate dallo stack di backtracking tutte le scelte per le clausole *p*.



## 5) La presenza del cut cosa va a mettere a rischio tra completezza e correttezza?

- Un uso scorretto del Cut mette a rischio la Completezza perché per snellire tagliamo rami ma se la soluzione stava in un ramo rimosso non la troveremo mai quindi, la completezza si viene a perdere. La correttezza rimane invariata, mica mette a rischio la correttezza delle risposte.
  In prolog abbiamo due stack uno di esecuzione (esecuzione dei predicati attivi in quel momento) e uno di backtracking (stack dei punti di backtracking che sono ancora da esplorare, cioè le alternative). Con il cut rendiamo definitive le scelte effettuate. Non ammettiamo backtracking nei predicati definiti prima del cut e quindi alteriamo il controllo del programma perdendo dichiaratività cioè: se usiamo il cut bisogna fare anche attenzione all'ordine con cui scriviamo fatti e regole perché poi il cut elimina le alternative, nel codice dobbiamo mettere quelle che vogliamo che vengano considerate prima.
  Dobbiamo quindi fare attenzione a come dichiararlo, diciamo che va per tutte quelle volte in cui non ci serve andare a scegliere alternative.

## 6) Perché in prolog non c'è la negazione classica?

- Proprio per non far saltare il discorso delle clausole di Horn. Perché in Prolog quando facciamo la derivazione SLD usiamo come metodo di prova la resolution che lavora per refutazione. Se ci fosse la negazione classica salterebbe tutto il meccanismo di base della resolution perché, invece di essere tutti negati gli elementi della formula, quelli che avevano già la negazione prima, sarebbero negati due volte e perciò tornerebbero positivi

## 7) Ragionamento non monotono

- Una forma di ragionamento si dice non monotono se non vale la monotonicità. Per monotonicità intendiamo che: a partire da una KB iniziale se riusciamo a inferire (dedurre) una certa formula F allora vale anche che la stessa formula F la riusciamo a inferire qualora la nostra KB la arricchissimo con altre informazioni. Quindi la domanda da porsi è: riusciamo a dedurre quello che deducevamo prima, dopo l'aggiunta di nuova conoscenza alla KB?

  Esempio: KB [i gatti miagolano; tom è un gatto]; F [Tom miagola]
  Nuova_formula [i gatti hanno i baffi] → tom continua a miagolare lo stesso anche dopo la nuova formula.
  La non-monotonicità si verifica quando non vale questa situazione.

  *Esempio "uccelli pinguini" se diciamo che tweety è un uccello e quindi vola, se diciamo che i pinguini sono uccelli ma se aggiungiamo una nuova informazione cioè che tweety è un pinguino rimane un uccello ma non vola più quindi non vale quanto valeva prima→non monotonicità*

Facendo così non creiamo inconsistenza perché aggiungiamo informazioni in più che fanno cambiare le deduzioni iniziali, prima so che tweety è un uccello allora dico che vola ma se poi mi dici che tweety è un pinguino allora so che è un uccello ma ritraggo il fatto che voli.
La logica classica è monotona, quindi il discorso uccelli e pinguini non lo possiamo fare, mentre in Prolog, con la negazione per fallimento, ci allontaniamo dalla logica classica e ci avviciniamo al ragionamento non monotono.

## 8) I 3 possibili esiti con negazione per fallimento

- sta domanda non ha senso, secondo me vuole sapere gli esiti di una dimostrazione SLD. che sono
  - Successo: troviamo un insieme vuoto
  - Fallimento finito: se non possiamo derivare il goal o comunque non riusciamo ad ottenere la clausola vuota
  - Fallimento infinito: se e' sempre possibile derivare nuove clausole risolventi, e quindi continuiamo a cercare una soluzione generando dei loop.

## 9) Perché e come la negazione per fallimento è associata al ragionamento non monotono?

- Perché la negazione per fallimento si allontano dalla logica classica che segue un ragionamento monotono. Per ragionamento monotono si intende che, a partire da un KB iniziale e da una formula F che vogliamo inferire, se ci aggiungessimo nuove info all'interno della nostra KB, riusciremmo comunque a dedurre F. Nel ragionamento non monotono questo non avviene, in quanto, tramite nuove informazioni all'interno della mia KB, non e' detto che posso dedurre lo stesso F (vedi esempio del pinguino domanda sopra). Pertanto, poiché' in Prolog ci allontaniamo dalla logica classica e abbiamo la negazione per fallimento, allora ci avviciniamo maggiormente al ragionamento di tipo non monotono.

```
campione(X):-
   topPlayer(X),
   \+sfotte(X).
topPlayer(neymar)
topPlayer(messi).
sfotte(neymar).
```

- Fare Esempio → in questa KB se andiamo a eseguire il goal "campione(messi) la risposta sarà True ma se andiamo ad aggiungere il fatto "sfotte(messi)" allo stesso goal la risposta sarà false. Questo viola il concetto del ragionamento monotono.

## 10) Ricerca nello spazio degli stati → strategie non informate/informate

- Per effettuare una ricerca nello spazio degli stati in Prolog è necessario prima definire il dominio, che sarà caratterizzato con: uno stato iniziale, un insieme di azioni, un insieme di obbiettivi/goal e il costo di ogni azione. Lo spazio degli stati e'

definito dallo stato iniziale più degli stati che raggiungiamo tramite l'applicazione delle azioni. L'insieme degli stati che vengono attraversati andando da uno stato all'altro verrà detto cammino e ogni cammino avrà un suo costo dipendente dalle azioni effettuate. Chiaramente una soluzione al problema sarà un cammino dallo stato iniziale a uno stato obiettivo. Particolarmente interessante sarà la soluzione ottima, ovvero il cammino che ha il costo minimo. In Prolog uno stato è rappresentabile come un fatto che viene piano piano cambiato dalle azioni. Ogni azione verrà realizzata tramite due regole distinte: applicabile(precondizione per l'azione) e trasforma(post condizione dopo che si è applicata l'azione). Parlando ora delle ricerche ne abbiamo di due tipi, quelle non informate e quelle informate. Quelle più semplici sono quelle non informate che non usano nessuna euristica, in particolare sono la ricerca in profondità, che punta a scendere il più possibili nell'albero; abbiamo poi la ricerca in ampiezza che punta a controllare un numero di strade più ampio senza andare troppo in profondità. le ricerca informate come detto usano delle euristiche, un esempio sono A* che viene calcolata da f(n) = g(n) + h(n) dove h e una funzione euristica e g e il costo di un cammino dallo stato iniziale al nodo n (il noto attuale). Un alternativa della ricerca con A* e IDA* che fa una ricerca in profondità sfruttando un euristica.

## 11) Negazione per Fallimento

- Predicato extra logico introdotto da Prolog la cui sintassi e' \+. L'idea della negazione per fallimento è che quando si presenta una negazione, Prolog cerca dimostra l'affermazione e dopo aver finito nega il risultato. Se l'interprete Prolog incontra \+ A, cerca di dimostrare A. Nel caso in cui non riesca a dimostrare A, allora l'interprete restituisce *false*, altrimenti restituisce *true.*

## 12) Differenza tra prolog e clingo

- Le differenze tra Prolog e ASP sono:
    - In ASP l'ordine dei letterali non conta; questo perche' non esiste un interprete che deve scorrere tutte le regole come succedeva in Prolog.
    - Prolog e' goal-directed, ASP no. ASP cerca un modello
    - ASP, non essendo goal-directed non puo' andare in loop perche' cerca solo delle istanziazioni possibili per le variabili
    - Prolo possiede il cut, ASP no.

## 13) Definizione ASP

- L'answer set programming(ASP) e' una metodologia di programmazione nata nel momento storico in cui si cercava di trovare una semantica per la negazione per fallimento. In seguito e' diventata qualcosa di piu', ovvero il fondamento di una nuova tecnica di programmazione. Cominciamo dicendo che nell'ASP quello che cambia e' l'approccio: nel Prolog la soluzione e' la prova e viene trovata andando a cercare regole che unifichino. Questo non succede nell'ASP in cui le soluzioni sono qualcosa di simile ai modelli della logica classica e non li andiamo a cercare con un procedimento di

unificazione guidato dal goal. Questo approccio molto diverso e' utile per andare a risolvere alcuni problemi combinatori come il soddisfacimento di vincoli. Questo perché nel Prolog asseriamo quello che sappiamo, mentre nell'ASP do dei veri e propri vincoli.

## 14) Negazione classica vs. per fallimento

- Particolare attenzione dobbiamo porla anche sugli operatori di negazione; la negazione classica e' più forte rispetto alla negazione per fallimento. Se abbiamo -p, sapremo che questa clausola e' vera solo se sappiamo esplicitamente che p = false. Nella negazione per fallimento verra' comunque restituito *true* anche nel caso in cui non si possiedano informazioni su *p*.

## 15) Ridotto di un programma teoria (di solito chiede un esempio pratico)

- Innanzi tutto dobbiamo dare una definizione al ridotto di un programma ASP. Per farlo introduciamo il ridotto P^S di un programma P rispetto a un insieme di atomi S. Dato l'insieme degli atomi S, il ridotto P^S viene determinato applicando due semplici regole
  - Rimuovendo per intero ogni regola il cui corpo contiene notL con L ∈ S
  - Rimuovendo tutti i restanti notX con X qualunque dalle restanti regole
- A questo punto per costruzione P^S non contiene atomi con negazione per fallimento, L'idea dell'algoritmo per determinare l'answer set e' che una volta stabilito S, e calcolato P S, se S e P S coincidono allora S e' un answer set per il programma P.

## 16) Nel ridotto c'è un solo answer set ?

- (Consiglio di spiegare cos'è un ridotto di un ASP e poi dare la risposta) No per forza, è possibile che ci sia anche più di un Answer Set.
- Non necessariamente, nel ridotto e' possibile avere situazione in cui ci siano più answer set (vedi esempio sulle slide)

## 17) Integrity constraints

- Una particolarità di ASP è che nelle regole non è obbligatorio che vi sia la testa. Quando questo non succede allora ho un integrity constraint. Con questo intendiamo dire che *: − a1, . . . , an* è inconsistente solo se *a1, . . . , an* sono tutti veri.

## 18) E possibili creare degli integrity constraint in prolog ?

- Un possibilità potrebbe essere quella di effettuare delle ricerche bloccando specifici sotto alberi che non rispettano determinate condizioni. In questo modo e come se stessimo creando dei vincoli. (riposta molto a caso)

## 19) Esempio ASP Pinguino

```
fly(X):-bird(X), not abnormal(X).
bird(X):-penguin(X).
abnormal(X):- penguin(X).
penguin(tux).
bird(tweety).
```

- ---
  - la risposta sarà

    $Answer : 1$
    $bird(tweety)\ bird(tux)\ abnormal(tux)\ fly(tweety)\ penguin(tux)$
    $SATISFIABLE$

- Quello che ci sta dicendo CLINGO è che abbiamo trovato un possibile modello che è soddisfacibile, e che le informazioni ricavabili sono bird(tweety), . . . Andando a scrivere il seguente programma

```
fly(X):-bird(X), not abnormal(X).
bird(X):-penguin(X).
abnormal(X):-penguin(X).
penguin(tux).
bird(tweety).
penguin(tweety).
```

- il risultato sarà

  $Answer : 1$
  $bird(tweety)\ bird(tux)\ abnormal(tux)\ abnormal(tweety)$
  $penguin(tux)\ penguin(tweety)$
  $SATISFIABLE$

- È successo quello che ci potremmo aspettare. Aggiungendo l'informazione che tweety è un pinguino, scopriamo le stesse cose di prima eccetto una: fly(tweety).

- Cambiamo ancora una cosa la problema

```
fly(X):-bird(X), not -fly(X).
bird(X):-penguin(X).
-fly(X):-penguin(X).
bird(tweety).
penguin(tux).
```

  $Answer : 1$
  $bird(tweety)\ penguin(tux)\ bird(tux)\ -fly(tux)\ fly(tweety)$
  $SATISFIABLE$

- Usando la negazione per fallimento seguita dalla negazione classica otteniamo un risultato(ovviamente) diverso. La particolarità in questo caso e' che non solo sappiamo che tweety vola, ma come risulta dalla risposta sappiamo anche che tux non vola!

## 20) Esempio ASP Nixon Pacifista-Quachero (lo ha chiesto spesso)

```
pacifist(X):- quaker(X), not -pacifist(X).
-pacifist(X) :- repubblican(X), not pacifist(X).
repubblican(nixon).
quaker(nixon).
```

- La prima regola ammettiamo che se una persona è quacchero e non riusciamo a reperire informazioni sul fatto che non sia pacifista, allora diremo che quella persona e' pacifista. La seconda regola invece dice che per un repubblicano, a meno che non si riesca a reperire esplicitamente il fatto che è pacifista, daremo per scontato che non sia un pacifista. La soluzione che troviamo con CLINGO è la seguente.

$$Answer: 1$$
$$repubblican(nixon)\ quaker(nixon)\ pacifist(nixon)$$
$$Answer: 2$$
$$repubblican(nixon)\ quaker(nixon)\ \ -pacifist(nixon)$$
$$SATISFIABLE$$

  - Ovvero abbiamo ottenuto che Nixon e' in due modelli diversi sia pacifista che non pacifista. In questo caso la soluzione e' dunque poco significativa. Infatti in generale, per essere vero qualcosa deve esserlo in tutti i modelli.
- Consideriamo adesso una cosa. In realtà' potremmo non usare esplicitamente la negazione classica ma sostituirla con qualcosa del tipo.

```
pacifist(X):- quaker(X), not nonpacifist(X).
nonpacifist(X) :- repubblican(X), not pacifist(X).
repubblican(nixon).
quaker(nixon).
```

- ```
  :-pacifist(X), nonpacifist(X).
  ```
  - Ovvero abbiamo definito un nuovo predicato *nonpacist*. Il predicato *nonpacist* è pero in relazione con il predicato pacist(uno è l'opposto dell'altro). Quindi questa situazione deve essere segnalata mediante l'integrity constraint: −pacif ist(X), nonpacifist(X).

## 21) Esempio
  ```

# DOMANDE MICALIZIO

## Planning Classico

### 1) Cos'è il planning in generale

- *"il planning e' l'arte e la pratica di pensare prima di agire"*. Il planning in generale e un processo deliberativo che sceglie ed organizza le azioni in base all'effetto che ci si aspetta queste producano, l'intelligenza artificiale studia la calcolabilità di questo processo deliberativo

### 2) Cosa è il planning e i tre tipi di algoritmi utilizzati per risolverlo (ricerca nello spazio dei piani, degli stati, ecc.…)

- In che senso ? Parlare di planning nello spazio dei piani o degli stati ?

### 3) STS (Definizione, STS-Grafo)

- Quando parliamo di planning dobbiamo assolutamente sviluppare un modello concettuale per poter utilizzare delle tecniche di ricerca. Creiamo quindi lo State Transition System Σ = (S, A, E, γ), Dove S rappresenta l'insieme finito di stati ricorsivamente enumerabili, A l'insieme finito di azioni applicabili ricorsivamente enumerabili, E l'insieme finito di eventi ricorsivamente numerabili e γ è una relazione di transizione di stato. Applicare un azione a uno stato causerà una transizione (γ) da s a s'. Un STS inoltre può essere rappresentato come un grafo diretto G = (N_g , E_G ), Dove N_g e l'insieme dei nodi de grafo e E_G è l'insieme di archi nel grafo.

## Problema di pianificazione Classica (+ Complessità PlanSAT, Bounded PlanSAT)

### 4) Come formalizzarlo, Assunzioni  e cosa si assume riguardo le osservazioni

- un problema di pianificazione classica è definito come **P = (Σ,s0, Sg),** dove **Σ** è un STS (guarda la domanda 3), **s0** ∈ S è lo stato iniziale, **Sg** ⊂ S è l'insieme degli stati goal. La soluzione del problema è una sequenza totalmente ordinata di azioni istanziate (ground) π = (a1, a2,..., an) che danno origine a una sequenza di transizioni di stato (s0,s1,...,sn).  Un buon algoritmo di pianificazione dovrebbe essere **Corretto** (si dice corretto se tutte le soluzioni che trova sono piani corretti), **Completo** (Un pianificatore è completo se trova una soluzione quando il problema è risolubile) e **Ottimo**(Un pianificatore è ottimo se l'ordine con cui le soluzioni sono trovate è coerente con una qualche misura di qualità dei piani). Un problema principale degli algoritmi di planning è che è dimostrabile che il Planning è un task computazionalmente costoso ricorrendo a due problemi decisionali, **PlanSAT**(che si chiede se esiste un piano che risolve un problema di pianificazione? ) e **Bounded PlanSAT**(che si chiede se esiste un piano di lunghezza k?). Per la pianificazione classica entrambi i problemi sono decidibili ma se estendiamo il linguaggio con simboli di funzione lo spazio di ricerca diventa infinito e **PlanSAT** diventa semi

decidibile(non e sempre possibile rispondere alla domanda) e **Bounded PlanSAT** rimane decidibile(avremo sempre una risposta). Inolte ce da segnalare che **Bounded PlanSAT** è NP Complesso mentre **PlanSAT** è P completo.

# Planning nello spazio degli stati

## 5) Progression e regression

- Parlando di algoritmi di pianificazione abbiamo molte scelte da prendere, La direzione della ricerca(**Progressione, regressione, bi-direzionale**), la rappresentazione dello spazio di ricerca(**spazio di ricerca esplicito** che coincide con lo spazio degli stati del transition system che modella il dominio oppure **spazio di ricerca simbolico** in cui ogni stato corrisponde a insiemi di stati del dominio)se usare un algoritmo di ricerca(**Non informato, informato o con euristiche locali**) e infine come controllare la ricerca(**Quindi usando euristiche specifiche per la ricerca informata o tecniche di prunnig**)

### a) Descrivere la strategia progression: come sono individuate le azioni che faranno parte del piano?

- Si procede dallo stato s0(stato iniziale) e usando per esempio dei meccanismi si means end analysis(che serve per capire quali azioni ci avvicinano al goal tramite dei meccanismi di valutazione come euristiche) si procede attivando le azioni che ci avvicinano al completamento dei goals.

### b) Spiegare la strategia regression: in cosa differisce dalla progression?

- Invece di partire dallo stato iniziale si parte dallo stato goal e si procede all'indietro cercando di capire quali sono le mosse migliori, Questo è anche detto meccanismo backward.

## 6) Ricerca nello spazio degli stati vs ricerca nello spazio dei piani (min 13:30 lez 3)

- Sta cosa no dovrebbe stare qua ma sotto

# STRIPS

## 7) Definizione (Linear Planning + Means-End Analysis)

- Strinps sta per STanford Research Institute Problem Solver. Introduce una rappresentazione esplicita degli operatori di pianificazione, fornisce una operazionalizzazione delle nozioni di differenza tra stati, subgoal, applicazione di un operatore e inoltre gestisce il frame problem. il tutto si basa su due idee fondamentali ovvero il **linear planning** e il **Means-End Analysis**. L'idea dietro il **linear planning** è risolvere un goal per volta e passare al successivo solo quando il precedente è stato raggiunto(quindi il meccanismo dividi e conquista); questo per diminuire la complessità del problema. Questo viene permesso dal fatto che Strips usa uno stack dei goals. Cosa comporta il linear planning ? → no interleaving sui goals, la ricerca è facilitata perché i goal non interagiscono tra loro Però proprio a causa della divisione dei goals

incorriamo nella Sussmann's Anomaly, ovvero che se abbiamo raggiunto un goal per raggiungere il successivo dobbiamo disfare il lavoro fatto. La **Means-End Analysis** considera due idee di base → cerca di considerare solamente gli aspetti rilevanti al problema (backward) e con quali mezzi (means, cioè operatori) sono disponibili e necessari per raggiungere il goal (end). Occorre quindi capire le differenze tra stato corrente e goal (means-ends analysis) dopodiché di trovare un operatore che riduce tale differenza dopodiché ripetere l'analisi sui sottogoal ottenuti per regressione attraverso l'operatore selezionato.

## 8) Stati, Relazioni, Plan Operators, Azioni, Applicabilità, Funzione gamma(s,a)

## 9) Algoritmo (+ Vantaggi e Svantaggi + Incompletezza e Correttezza)

- l'algoritmo prende uno stato iniziale, inizializza la lista che conterrà i passi del piano e inizializza uno stack dei goals. Si pushano i goals sullo stack e si inizia a eseguire il ciclo fino a quando lo stack non è vuoto. Il ciclo è il seguente →
  - Se il goal in cima allo stack è completato lo si rimuove
  - Se il goal in cima allo stato è un goal congiunto (più goal) lo si divide in sotto goal, si decide un ordine e si pushano sullo stack i sotto-goals.
  - Se il goal in cima allo stack c'è un goal semplice allora
    - scegliere un operatore che ha l'effetto di creare un match con il goal
    - si rimpiazza il goal e con l'operatore
    - e si pusha la condizione in cima allo stack
  - Se in cima allo stack abbiamo un operatore
    - applichiamo l'operatore allo stato
    - aggiungiamo l'operazione al piano
- Svantaggi → usando un linear planning
  - linear planning può produrre piani subottimi (se abbiamo goals sub ottimi)
  - linear planning è incompleto
  - Si può creare l'anomalia di sussman
- Vantaggi → usando un linear planning
  - spazio di ricerca ridotto grazie al linear planning
  - Ottimo con goal che sono simple goal(non congiunti tra loro)
  - è sound → rischio di non troviamo una soluzione anche se esiste

## 10) Anomalia di Sussman

- L'anomalia di Sussman deriva dal passo di Linear Planning di Strips, la risoluzione di un singolo goal alla volta crea problemi con goal che vanno considerati in congiunzione tra loro(goal dipendenti tra loro). A causa di questa anomalia una volta completato uno dei due goals dovremo disfare tutto il lavoro fatto per completare il secondo.

### 11) Oltre all'anomalia di Sussman, quali altri problemi ha STRIPS?

- linear planning può produrre piani sub-ottimi (se consideriamo la lunghezza il parametro di qualità)
- linear planning è incompleto

# Planning nello spazio dei piani

## 12) PSP

#### a) Perché è un pianificatore particolare, in quale spazio cerca questo pianificatore? **(domanda frequente)**

- l'idea che abbiamo inizialmente è quella di formulare un problema di pianificazione come un problema di soddisfacimento di vincoli (CSP), usare un CSP consente di ritardare alcune decisioni fino a quando non è strettamente necessario.Questo principio diventa quindi un fondamento del plannig e verrà chiamato **Least-Commitment Planning.** Questo principio dice di fare scelte solo quando strettamente indispensabile per risolvere un problema (Possiamo ritardare Ordinamenti e Bindings ovvero il vincolare le variabili) e di evitare di porre più vincoli del dovuto durante la ricerca → questo principio ci portare a sviluppare tutta l'albero di ricerca in cui ogni dono sarà un piano parzialmente ordinato con difetti, a ogni passo rimuoveremo dei difetti/Flaws raffinando il piano. Una volta che l'algoritmo termina avremo un piano istanziato ma parzialmente ordinato.

#### b) Cosa indica il principio Least-Committment? Qual è l'idea di funzionamento presente alla base del Least-Commitment Planning?

- Questo principio dice di *fare scelte solo quando strettamente indispensabile per risolvere un problema* (**Possiamo ritardare Ordinamenti e Bindings ovvero il vincolare le variabili**) e di *evitare di porre più vincoli del dovuto durante la ricerca* → questo principio ci portare a sviluppare tutta l'albero di ricerca in cui ogni dono sarà un piano parzialmente ordinato con difetti, a ogni passo rimuoveremo dei difetti raffinando il piano. Una volta che l'algoritmo termina avremo un piano istanziato ma parzialmente ordinato.

#### c) State-Variable Representation (Stato, Applicabilità, $\gamma(s,a)$)

- Non capisco a cosa si riferisce

#### d) Proprietà degli oggetti

- Nelle slide non ci sono riferimenti a questa domanda

#### e) Piano <A,O,B,L>

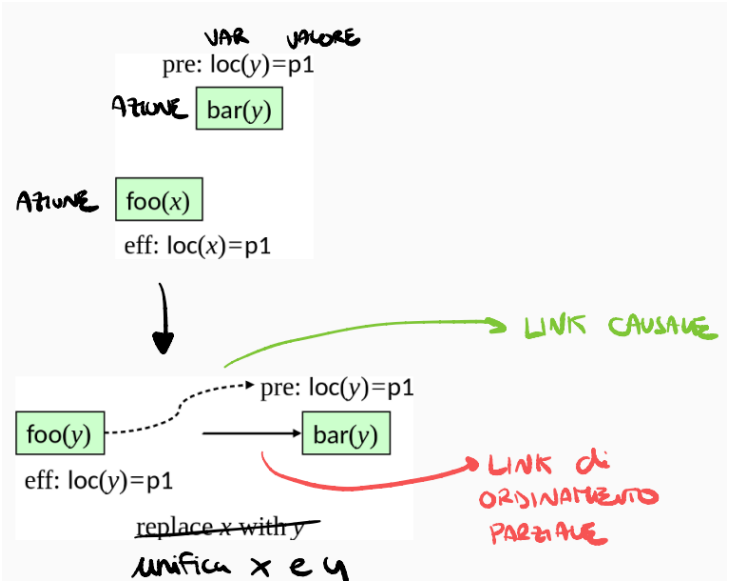
- Ogni nodo del grafo è un Piano, questo piano è rapresentato come una tupla <A,O,B,L> dove:
    - A è un insieme di azioni anche solo parzialmente istanziate
        - a0 è l'azione iniziale

        - a_inf equivale a un azione senza effetto che sarebbe quindi una precondizione per uno stato goal
    - O insieme di vincoli di ordinamento della forma (a_i < a_j), è una relazione d'ordine solo parzialmente definita
    - B è un insieme di "bindings" (vincoli)
        - I vincoli si riferiscono al possibile assegnamento delle variabili che possono essere uguali o diverse da costanti, oppure uguali o diverse da altre variabili.
    - L è un insieme di causal links(link causali) della forma a_i –c→ a_j a indicare che c è un effetto dell'azione a_i che è necessario (cioè precondizione) per l'azione aj. c è un'assegnamento di un valore a una variabile. → quindi a_i genera la precondizione che si attivi l'azione a_j trammite c.

### f) Causal link?

- I casual link o link casuali sono l'elemento L della quadrupla <A,O,B,L> che rappresenta un nodo ovvero un piano del grafo. L è un insieme di causal links(link causali) della forma a_i –c→ a_j a indicare che c è un effetto dell'azione a_i che è necessario (cioè precondizione) per l'azione aj. c è un'assegnamento di un valore a una variabile. → quindi a_i genera la precondizione che si attivi l'azione a_j trammite c.

### g) Flaws (Open Goals, Threats) / Quali sono i possibili flaws e quali i loro corrispondenti resolvers?

- Abbiamo detto che ogni nodo del grafo e un piano parzialmente ordinato con difetti/Flaws. È l'algoritmo eliminerà i vari difetti lasciando solo un piano istanziato ma parzialmente ordinato. Abbiamo principalmente due tipi di difetti:
    - Open goals: Abbiamo un'azione B che richiede una precondizione P, ma P non è una conseguenza di nessuna Azione. per risolvere questa situazione ci servono due cose, O avere un azione A che genera P, o avere un azione A che precede B. in ogi caso dovremo prendere questa A e creare il link causale con B passando per P (A –P→B).
    - 

- Threats: il secondo problema da risolvere è quando abbiamo più link causali sulla stessa azione. Abbimo 3 azioni: A, B e C e A e C hanno un link causale su B(ovvero la loro post condizione e la precondizione di B). per risolvere questo problema abbiamo 3 modi.
  - **Promotion:** Imporre che C precede A
  - **Demotion:** Imporre che B precede B
  - **Separation:** implore un vincolo di *non considerazione* in modo tale che l'effetto di C non unifichi con p(ovvero che eseguendo C non causiamo p che è la preconzione di B)

### h) Correttezza e Completezza

- L'algoritmo è corretto e completo, ritorna un piano parzialmente ordinato π tale che qualsiasi ordinamento totale delle sue azioni raggiunge il goal. Dove il dominio lo consenta, le azioni non strettamente sequenziali possono essere eseguite in parallelo.

### i) Cosa vuol dire regredire un goal? Quando si parla di stato e quando di parla di goal?

- Extract-Solution invoca GP-Search su un livello di azioni e su un sottogoal (regredito dal goal del problema)

### 13)

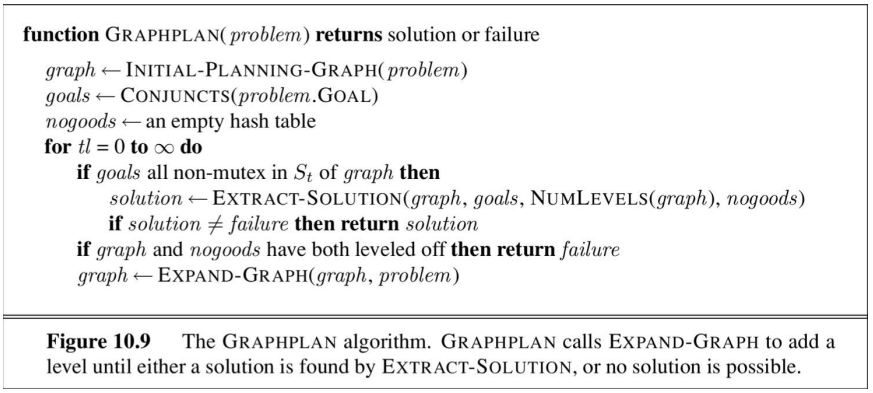
# Graph Plan

## 14) GraphPlan (cos'è un grafo di pianificazione, cosa dice il thm, come funziona a grandi linee l'algoritmo)

- Un grafo di pianificazione e' una struttura dati utile per definire euristiche domain-independent e generare un piano (graphplan). Il grafo astrae lo spazio di ricerca e può essere costruito con un algoritmo polinomiale.
  Un grafo di Pianificazione (GP) e' un grafo aciclico, orientato e organizzato a livelli, in cui:
  - Il livello iniziale S0 contiene i letterali che valgono nello stato iniziale (un nodo per ogni fluente)
  - Il livello di azioni A0 contiene le azioni ground che possono essere applicate ad S0 (un nodo per ogni azione)
  - i due livelli S*i* e A*i* si alternano fino a quando non si raggiunge una condizioni di terminazione tale che:
    - S*i* contiene tutti i letterali che *potrebbero* valere al tempo *i*
    - A*i* contiene tutte le azioni che *potrebbero* avere le precondizioni soddisfatte al tempo *i*.

  Graphplan si basa sulla seguente teoria: *Se esiste un piano valido, allora questo e' un sottografo del grafo di pianificazione.*
  Si suddivide in diversi passi:
  1. Espandi il grafo un livello per volta fino a quando tutti gli atomi del goal non compaiono all'interno dell'i-esimo stato

2. invoca EXTRACT-SOLUTION per cercare un piano all'interno del grafo:
    a. se EXTRACT-SOLUTION trova una soluzione, termina con successo
    b. altrimenti, termina insuccesso
    c. altrimenti, vai al passo 3
3. espandi il grafo ancora di un livello e torna al passo 2.



```
function GRAPHPLAN(problem) returns solution or failure
  graph ← INITIAL-PLANNING-GRAPH(problem)
  goals ← CONJUNCTS(problem.GOAL)
  nogoods ← an empty hash table
  for tl = 0 to ∞ do
    if goals all non-mutex in S_t of graph then
      solution ← EXTRACT-SOLUTION(graph, goals, NUMLEVELS(graph), nogoods)
      if solution ≠ failure then return solution
    if graph and nogoods have both leveled off then return failure
    graph ← EXPAND-GRAPH(graph, problem)
```

**Figure 10.9** The GRAPHPLAN algorithm. GRAPHPLAN calls EXPAND-GRAPH to add a level until either a solution is found by EXTRACT-SOLUTION, or no solution is possible.

Dallo pseudocodice vediamo che abbiamo un grafo con un planning iniziale per un certo problema e un goals che e' determinato dalla congiunzione di più goal per quel determinato problema. Abbiamo poi la variabile *nogoods* che rappresenta una coppia <g, livello> e rappresenta il fatto che ho tentato di risolvere il goal *g* nel livello e non ci sono riuscito; questo mi fa capire che questo goal non potrà essere risolvibile in tutti i livelli precedenti, ma potrebbe esserlo a tutti i livelli successivi. EXTRACT-SOLUTION effettua una ricerca partendo dal goal all'interno del grafo di pianificazione ed invoca un'altra funzione GP-SEARCH che pianifica per il solo livello su cui e' invocato e, se ha successo, invoca EXTRACT-SOLUTION sul livello di stato. Pertanto se EXTRACT-SOLUTION restituisce una soluzione che non sia *failure,* allora restituisce una soluzione. Inoltre se il grafo e l'insieme dei nogoods e' stato livellato (non posso più espandermi ulteriormente), allora restituisco *failure*, altrimenti espando il grafo e continuo a cercare una soluzione.

Possiamo concludere che EXTRACT-SOLUTION e' un algoritmo di backward di ricerca all'interno del grafo di pianificazione. Ogni volta che EXTRACT-SOLUTION applicata a certi letterali L e ad un certo livello l fallisce, allora la coppia <L, l > viene aggiunta all'insieme dei nogoods

## 15) Dimostrare la terminazione di GRAPHPLAN

- Abbiamo visto che l'algoritmo termina quando si raggiunge una soluzione o se non è più possibile espandere ulteriormente il grafo. *Ma fino a quando e' possiibile continuare ad espandere il grado dopo che si e' livellato?*

Qui entrano in gioco i nogoods perché questi decrescono; se EXTRACT-SOLUTION non trova una soluzione, deve esistere un sottoinsieme di atomi del goal che, essendo non raggiungibili, vengono marcati come nogoods. Allora se e' possibile che vi siano meno nogoods al livello successivo, dobbiamo continuare ad espandere il grafo. Affinché il criterio di terminazione sia valido, dobbiamo dimostrare che:

- i nogoods si livellano sempre
- quando i nogoods e il grafo sono livellato, allora la soluzione esiste.

Per dimostrare che i nogoods si livellano sempre, dobbiamo basarci su delle proprietà monotone del grafo di pianificazione in cui:

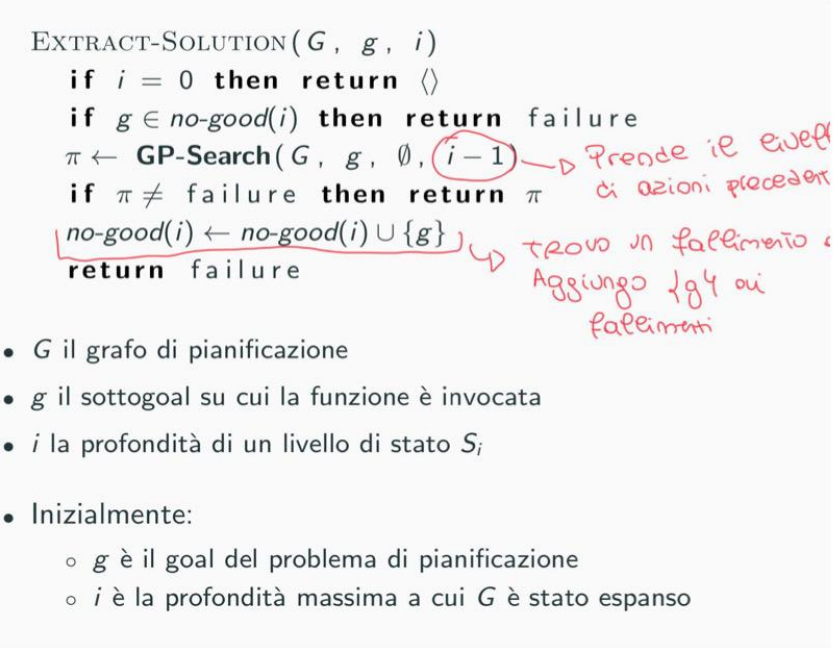
- i letterali crescono monotonicamente: una volta che un letterale compare in un livello, sarà sempre presente in tutti i livelli successivi
- le azioni crescono monotonicamente: una volta che un'azione compare in un livello, sarà sempre presente in tutti i livelli successivi
- le relazioni di mutex decrescono monotonicamente: se due azioni sono in mutec in un livello, allora lo sono anche in tutti i livelli precedenti
- i nogoods decrescono monotonicamente: se un sottoinsieme di atomi di goal non e' raggiungibile in un certo livello, allora non lo sarà nemmeno nei livelli precedenti.

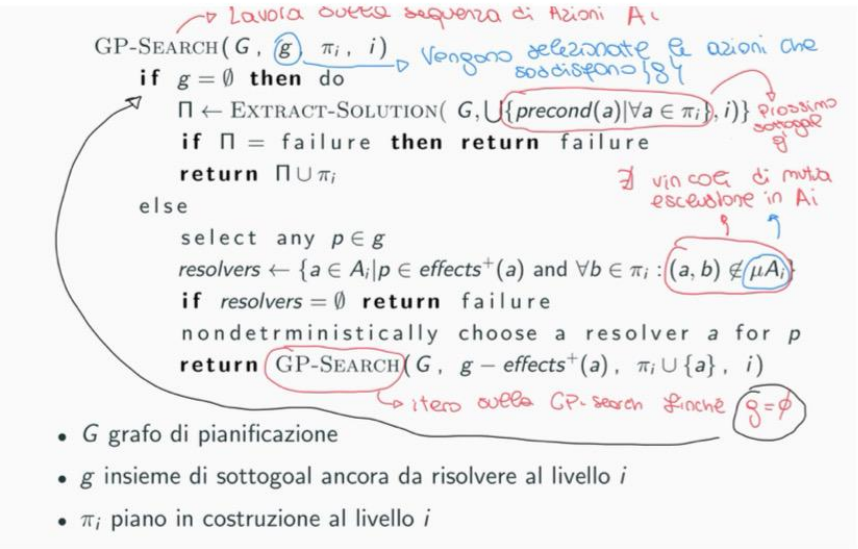
Per quanto riguarda invece la dimostrazione che i nogoods e il grafo si livellano sempre abbiamo:

- dato che le azioni e i letterali crescono monotonicamente, e dato che sono entrambi insiemi finiti, deve esistere un livello in cui l'insieme dei letterali e' uguale a quello dei letterali del livello precedente.
- poiché i mutex e nogoods decrescono monotonicamente e poiché non possono esserci meno di zero mutex e nogoods, deve esistere un livello con lo stesso numero di mutex e nogoods del precedente
- una volta che il grafo ha raggiunto la situazione in cui sia i livelli che i nogoods si sono livellati, se uno dei letterali del goal e' mancante o in mutex con un altro letterale del goal, allora si può concludere che non esiste una soluzione.

## 16) Come si comportano GP-Search e Extract-Solution)? cosa è un no-good?

- Vediamo una rappresentazione algoritmica di entrambi:




EXTRACT-SOLUTION e' un algoritmo di backward di ricerca all'interno del grafo di pianificazione, considerandolo come una sorta di grafo AND/OR. Effettua una ricerca partendo dai goal(backward) . Prende in input Il grafo di pianificazione G, il sottogoal *g* su cui la funzione e' invocata e la profondita' *i* di un livello di stato *Si*. Inizialmente ho che la profondità e' nulla, allora restituisco un insieme vuoto <>. Successivamente se il sottogoal appartiene all'insieme dei nogoods, restituisce failure. EXTRACT-SOLUTION invoca poi

una funzione GP-SEARCH che pianifica per il solo livello su cui e' invocato e, se ha successo, invoca EXTRACT-SOLUTION sul livello di stato. Se il piano non fallisce, restituisce il piano stesso; se trovo un fallimento aggiungo *g* all'insieme dei nogoods e termino come *failure.*

GP-SEARCH lavora sulle sequenze di azioni A*i*, in cui in input prende il grafo G, l'insieme delle azioni che soddisfano *g*, il piano di costruzione *pigreco* al livello *i.* Nella GP-SEARCH, se non ho un sottogoal a disposizione, genero un nuovo piano tale che richiamo EXTRACT-SOLUTION per *g'*. Se Il piano ∏ = *failure,* allora restituisce *failure* e restituisco l'insieme dei piani ∏ a cui unisco il piano di costruzione *pigreco* al livello *i.* Altrimenti, dato un un sottogoal *p*, creo un *resolvers* per un'azione *a* tale che *p* appartiene agli effetti di *a* e per ogni piano *b* ho che l'azione *a* nel piano *b* appartiene all'insieme delle azioni A*i.* Restituisco la funzione di GP-SEARCH con i valori aggiornati per *a*.

## 17) Grafo dei piani (Livelli, Costruzione del grafo, Livella, Mutex, Complessità e Raggiungibilità del goal)

- Un grafo di pianificazione e' una struttura dati utile per definire euristiche domain-independent e generare un piano (graphplan). Il grafo astrae lo spazio di ricerca e può essere costruito con un algoritmo polinomiale.
  Un grafo di Pianificazione (GP) e' un grafo aciclico, orientato e organizzato a livelli, in cui:
  - Il livello iniziale S0 contiene i letterali che valgono nello stato iniziale (un nodo per ogni fluente)
  - Il livello di azioni A0 contiene le azioni ground che possono essere applicate ad S0 (un nodo per ogni azione)
  - i due livelli S*i* e A*i* si alternano fino a quando non si raggiunge una condizioni di terminazione tale che:
    - S*i* contiene tutti i letterali che *potrebbero* valere al tempo *i*
    - A*i* contiene tutte le azioni che *potrebbero* avere le precondizioni soddisfatte al tempo *i*.

  Nella **costruzione del grafo** occorre tenere conto che:
  - un letterale al tempo *i* può essere inteso sia come *precondizione* di un azione A*i* sia come atomo persistente
  - in ogni livello S*i* possono tenere presenti link di mutua esclusione tra letterali
  - In ogni livello A*i* possono tenere presenti link di mutua esclusione tra azioni
  - GP cresce monotonicamente: due stati consecutivi Si e Si+1 sono identici. Questo accade solo se il grafo e' livellato, ovvero quando ho determinato l'insieme di tuti i possibili letterali partendo da S0 applicando in tutti i possibili modi le azioni:
    - ogni livello Ai contiene tutte le azioni che potrebbero essere applicate al tempo *i*
    - ogni livello Si contiene tutti i letterali applicabili al tempo *i*

- i vincoli di mutex indicano quali letterali o azioni esistere simultaneamente.

Parliamo adesso del **vincolo di mutua esclusione tra due azioni** di un livello A*i*. Tale vincolo tra due azioni viene messo se mi trovo davanti a una delle seguenti condizioni:

- **Effetti inconsistenti**→ gli effetti di un'azione negano gli effetti dell'altra
- **Interferenza**→ quando le precondizioni di un azione sono mutuamente esclusive con gli effetti di un'altra azione.
- **Competizione delle precondizioni** → una delle precondizioni di un'azione e' mutuamente esclusiva con le precondizioni dell'altra.

Per la **mutua esclusione tra due letterali** in un livello Si:

- Complementari → se uno e' la negazione dell'altro
- Inconsistent Support → se ogni possibili coppia di azioni al livello Ai-1 che producono due letterali sono mutuamente esclusivi.

Per stimare il **costo di una congiunzione di letterali** ci sono diverse euristiche:

- **Max Level** → prende il massimo livello in cui compaiono tutti i letterali del goal, anche con possibili vincoli di mutex
- **Somma di livelli** → assume indipendenza dei letterali del goal non e' ammissibile
- **Livelli di insieme** → profondità del livello in cui tutti i letterali del goal compaiono senza che alcuna coppia di essi sia in mutex. Funziona bene quando i letterali del goal non sono indipendenti tra loro.

## 18) Quando si smette di costruire il grafo?

- Si smette di costruire quando EXTRACT-SOLUTION cerca una soluzione e termina con successo, in alternativa quando sia il grafo che i nogoods si sono livellati senza che una soluzione sia trovata. Affinché il criterio di terminazione sia utilizzabile, bisogna dimostrare che:
  - i nogoods si livellano sempre
  - quando i nogoods e il grafo si sono livellati, allora la soluzione esiste.

## 19) Euristiche per stimare il costo della congiunzione di letterali

- Per stimare il **costo di una congiunzione di letterali** ci sono diverse euristiche:
  - **Max Level** → prende il massimo livello in cui compaiono tutti i letterali del goal, anche con possibili vincoli di mutex
  - **Somma di livelli** → assume indipendenza dei letterali del goal non e' ammissibile

- **Livelli di insieme** → profondità del livello in cui tutti i letterali del goal compaiono senza che alcuna coppia di essi sia in mutex. Funziona bene quando i letterali del goal non sono indipendenti tra loro.

## 20) Qual è il tempo di costruzione del grafo? Perché è polinomiale?

- Il tempo di costruzione del grafo e' O(*n (a + l) ^2 ).* Dati un numero di letterali *l* e un numero di azioni *a,* ogni livello Si ha non piu di *l* nodi e *l^2* link di mutex tra letterali. Ogni livello Ai non ha piu di *l + a* nodi (inclusi i no-op) e (*a + l)^2* link di mutex.
  Quindi un grafo con *n* livelli ha dimensione O(*n (a + l) ^2 )* e la stesso complessita' per costruirlo.

## 21) A cosa serve e a cosa può essere utile la struttura del grafo nella piafinicazione?

- Serve per:
  - definire strutture domain-independent
  - Generare un piano per un insieme di azioni a partire da uno stato S.

## 22) Cosa vuol dire che un grafo si `e livellato? Quando posso escludere la presenza di una soluzione solo guardando il grafo?

- Il grafo e' livellato quando ho determinato l'insieme di tuti i possibili letterali partendo da S0 applicando in tutti i possibili modi le azioni:
  - ogni livello Ai contiene tutte le azioni che potrebbero essere applicate al tempo *i*
  - ogni livello Si contiene tutti i letterali applicabili al tempo *i*
  - i vincoli di mutex indicano quali letterali o azioni esistere simultaneamente.

  Posso escludere la presenza di una soluzione quando i nogoods e il grafo si sono livellati senza trovare una soluzione.

## 23) Vincoli di mutua esclusione tra le azioni

- Parliamo adesso del **vincolo di mutua esclusione tra due azioni** di un livello A*i*. Tale vincolo tra due azioni viene messo se mi trovo davanti a una delle seguenti condizioni:
  - **Effetti inconsistenti**→ gli effetti di un'azione negano gli effetti dell'altra
  - **Interferenza**→ quando le precondizioni di un azione sono mutuamente esclusive con gli effetti di un'altra azione.
  - **Competizione delle precondizioni** → una delle precondizioni di un'azione e' mutuamente esclusiva con le precondizioni dell'altra.

# Domande Varie

## 24) Quali aspetti caratterizzano un algoritmo di pianificazione? Quali scelte vanno fatte nel momento in cui si pensa a un nuovo algoritmo di planning? →**Domanda che riguarda anche algoritmi di ricerca nello spazio dei piani**

- Parlando di algoritmi di pianificazione abbiamo molte scelte da prendere, La direzione della ricerca(**Progressione, regressione, bi-direzionale**), la rappresentazione dello spazio di ricerca(**spazio di ricerca esplicito** che coincide con lo spazio degli stati del transition system che modella il dominio oppure **spazio di ricerca simbolico** in cui ogni stato corrisponde a insiemi di stati del dominio)se usare un algoritmo di ricerca(**Non informato, informato o con euristiche locali**) e infine come controllare la ricerca(**Quindi usando euristiche specifiche per la ricerca informata o tecniche di prunnig**)

## 25) Parlare di qualche euristica

- La ricerca euristica è l'approccio più comune per la risoluzione di problemi di pianificazione, Infatti, molti pianificatori consistono di: algoritmo di ricerca + funzione euristica, l'euristica cerca di focalizzare la ricerca sul goal guidando le scelte non deterministiche. Una funzione euristica stima la "bontà" di una scelta in base alla direzione. Se andiamo in avanti/progression (dallo stato iniziale al goal) la scelta è il "next state" s stima come distanza da s al goal G. Se invece stiamo andiando all'indietro/regression/backward  la sc paelta è il "next sub-goal" sg: stima come distanza da sg allo stato iniziale. Ma come posso costruire una funzione euristica ? → ci sono diversi punti da seguire:
    - Definire un problema analogo all'originale ma rilassato (i.e., relaxed), cioè semplificato
    - Risolvere il problema rilassato (in maniera ottimale, se possibile)
    - Usare il costo della soluzione per il problema rilassato come euristica nel problema originale
- Ci sono diverse osservazioni da fare, La funzione euristica viene invocata per ogni stato dello spazio di ricerca considerato dall'algoritmo, deve quindi essere efficiente. Inoltre l'euristica sarà tanto più affidabile quanto il problema rilassato sarà vicino al problema reale, d'altra parte tenderà a essere più costosa. Le tecniche di rilassamento sono:
    - Delete relaxation: cancellazione degli effetti negativi dalle azioni
    - Astrazione: considerare un problema pi`u piccolo  (tipo il gioco dell'8 invece del gioco del 15)
    - Landmark: è un insieme di azioni (o fatti) t.c. Ogni soluzione deve avere almeno un'azione di quest'insieme, sono usati per definire euristiche che fanno un safe pruning dello spazio di ricerca.
- Per risolvere un problema rilassato ci sono più modi, come non risolvere il problema ma stimare solamente il costo di soluzione, risolvere in modo sub ottimo un problema o risolvere il problema ma in modo ottimo.

## 26) Spiegare come costruire delle euristiche non ammissibili basate sulla distanza sia per ricerche forward che per ricerche backward

- Non si fa riferimento a una risposate sulle slide

## 27) Se non volessimo un'euristica ammissibile che faremmo?

- (Questa risposta è una mia opinione) → per avere un euristica ammissibile innanzi tutto bisogna aver ben definito il dominio su cui andremo a operare, fatto ciò si possono vedere se esistono euristiche note per risolvere il problema. In caso di assenza di euristiche la si prende un dominio *rilassato* del problema che è più facile da risolvere, a questo punto di sviluppa un euristica e si testa sul dominio rilassato, se questa euristica da buoni risultati dovrebbe essere utilizzabile anche sul dominio che vogliamo risolvere. **Extra:** a sistemi intelligenti ci è stato spiegato che per valutare diverse euristiche su un problema bisogna effettuare diversi test sul problema e problema rilassato, facendo poi la medie dei risultati e capendo quale sia l'euristica migliore. Se abbiamo due euristiche A e B e A si comporta meglio di B allora possiamo dire che A domina b sul come euristica sul problema in questine.

## 28) Problema di pianificazione classico e come formalizzarlo, le sue assunzioni, che cosa si assume riguardo le osservazioni

- Un problema di pianificazione classica è definito come **P = (Σ,s0, Sg),** dove **Σ** è un STS, **s0** ∈ S è lo stato iniziale, **Sg** ⊂ S è l'insieme degli stati goal. La soluzione del problema è una sequenza totalmente ordinata di azioni istanziate (ground) π = (a1, a2,..., an) che danno origine a una sequenza di transizioni di stato (s0,s1,...,sn). Un buon algoritmo di pianificazione dovrebbe essere **Corretto** (si dice corretto se tutte le soluzioni che trova sono piani corretti), **Completo** (Un pianificatore è completo se trova una soluzione quando il problema è risolubile) e **Ottimo**(Un pianificatore è ottimo se l'ordine con cui le soluzioni sono trovate è coerente con una qualche misura di qualità dei piani). Un problema principale degli algoritmi di planning è che è dimostrabile che il Planning è un task computazionalmente costoso ricorrendo a due problemi decisionali, **PlanSAT**(che si chiede se esiste un piano che risolve un problema di pianificazione? ) e **Bounded PlanSAT**(che si chiede se esiste un piano di lunghezza k?). Quando parliamo di pianificazione classica abbiamo diverse assunzioni da rispettare, vediamone una lista e come rilassare i vincoli:
    - **0)Domini finiti / stati finiti**: abbiamo un numero finito di stati, se volessimo *rilassarlo* potremmo avere azioni che creano nuovi oggetti o gestire fluenti
    - **1) Completa osservabilità:** ovvero il dominio deve essere completamente osservabile da ogni punto di vista (per questo facciamo sempre domini giocatolo). Se volessimo *rilassarlo* dovremmo avere stati sconsciuti e non osservabili.
    - **2) Domini deterministici:** l'STS (Σ) è deterministico, se volessimo *rilassarlo* potremmo creare delle transizioni che portano a stati sconosciuti
    - **3) Dominio statico:** mentre computiamo un piano valido il mondo non cambia, se volessimo *rilassare* potremmo prendere domini dinamici

    - **4)Goal Semplici:** uno stato o un insieme di stati da raggiungere, se volessimo *rilassare* potremmo aggiungere goal complessi che aumentano il costo computazionale
    - **5)Piani sequenziali:** il piano è una sequenza finita di azioni che vanno eseguite una alla volta, se volessimo *rilassare* potremmo dare la possibilità di eseguire più azioni in parallelo o non introdurre vincoli che sono del dominio.
    - **6)Tempo implicito:** Le azioni e gli eventi non hanno durata, o in altri termini, hanno tutti durata istantanea, se volessimo *rilassare* potremmo trattare azioni durative i cui effetti si sviluppano nel tempo.
    - **7)Offline Planning:** Σ non cambia mentre il pianificatore sta inferendo un piano
    - **8)Single agent:**Un solo pianificatore e un solo controller (esecutore), in pratica un solo agente alla volta, se volessimo *rilassare* potremmo domini multi agente

## 29) Cosa succede se si usa la “modify” in un fatto?

- Modifico un fatto. Quando modifico un fatto, ne aggiorno l’indice. Pertanto equivale a rimuovere il fatto originale e aggiungere il nuovo fatto modificato con un indice incrementato.

## 30) Discorso generale sul problema della pianificazione (quindi cos'è, che elementi si usano, STS)

- Per pianificazione, intendiamo un insieme di azioni che, a partire da uno stato iniziale, ci portano al goal. Quando vogliamo modellare un modello di pianificazione, abbiamo bisogno di introdurre:
    - STS (State Transition System) → e’ un grafo diretto G= (Ng, Eg) dove:
        - Ng=S corrisponde all’insieme di tutti i possibili sottoinsieme di S, compreso l’insieme vuoto
        - Eg corrisponde all’insieme degli archi del grafo tale che esiste un arco che va da s ad s’ etichettato
    - Stabilire cos’e’ un **piano** → sequenza di azioni per raggiungere un determinato goal G.
    - Definire un **goal**, che può essere di diversa tipologia

## 31) Cosa si puo dire nel classical planning sulle precondizioni e sulle postcondizioni di un'azione

## 32) Pianificazione classica, quale algoritmo viola qualche vincolo della pianificazione classica?

## 33) Algoritmi backward

- Gli algoritmi backward o regression, partono dal goal e vanno “all’indietro”, il tutto utilizzando delle euristiche che passo per passo trovano il sotto goal migliore.

## 34) Strips tutto tranne codice algoritmo

- Leggere domande su Strips

## 35) Euristica Hg

- Quest’anno quelle slide non si devono fare

### 36) Euristiche domain Indipendent (tutte comprese GP e LCP)

- Secondo me la domanda è formulata male, GraphPlan serve per definire euristiche domain-independent (non dà garanzie sulla raggiungibilità di uno stato, ma fornisce una buona stima della distanza)
- Non capisco a cosa si riferisca con LCP

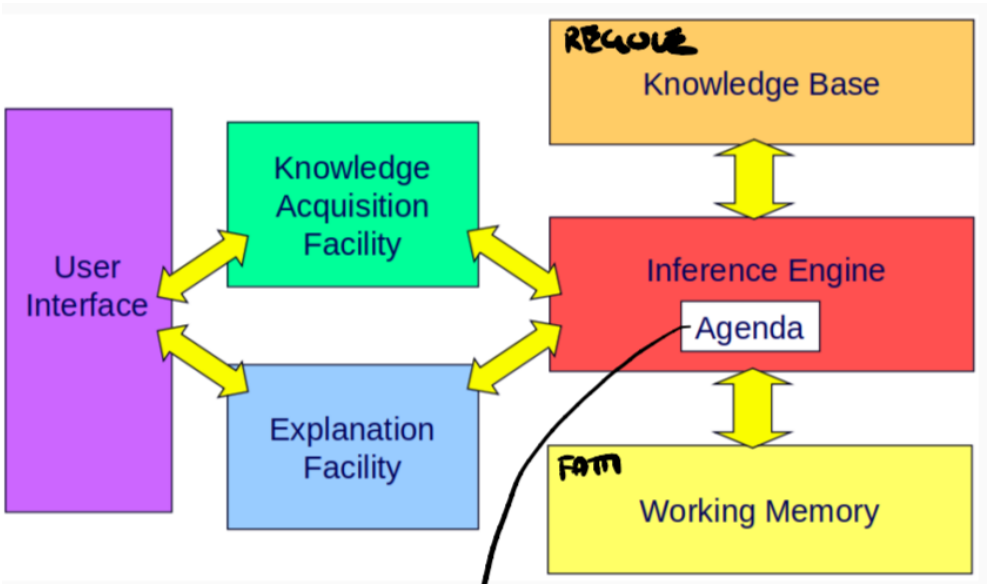
# Sistemi Esperti

### 37) Cosa è un Sistema Esperto?, quale aspetto lo caratterizza e quando quindi ha senso ricorrere a uno di questi e quando invece è meglio non usarlo?

- Un sistema esperto e un sistema basato sulla conoscenza `e un sistema in grado di risolvere problemi in un dominio limitato ma con prestazioni simili a quelle di un esperto umano del dominio stesso. Generalmente esamina un largo numero di possibilità e costruisce dinamicamente una soluzione.
- Un sistema esperto non `e un insieme d'istruzioni immutabili che rappresentano la soluzione del problema ma e un ambiente in cui viene rappresentata la conoscenza, si ragiona su questa conoscenza per effettuare dei task e si modifica questa conoscenza, ovvero si aggiunge rimuove. Un sistema e caratterizzato inoltre da Specificità (conoscenza di dominio), Rappresentazione esplicita della conoscenza. Meccanismi di ragionamento. Capacità di spiegazione e Capacita di operare in domini poco strutturati
- Non ha senso ricorre a un sistema esperto quando → esistono algoritmi ”tradizionali” efficienti per risolvere il problema considerato, l’aspetto principale del problema `e la computazione e non la conoscenza,  la conoscenza non può essere modellata o esplicitata in modo efficiente e infine quando l’utente finale `e riluttante ad applicare un sistema esperto per via della criticità del task

#### a) Quali sono i componenti fondamentali di un SE

- Un sistema esperto e caratterizzato dalle seguenti componenti → **Knowledge-Base [KB] (Rule-Base)**: mantiene la conoscenza dell’esperto come regole condizione-azione (aka: if-then or premise-consequence). Una **working memory** che mantiene i fatti iniziali e quelli generati dalle inferenze. Un **inference Engine** che al il compito di fare → pattern matching(per attivaszioni if delle regole), gestire l’agenda(ovvero decidere quali regole attivare prima) e un **Execution**(esecuzione della regola in cima all’agenda). **Explanation facility** che fornisce una giustificazione delle soluzioni. Un **Knoledge Acqusition facility** che aiuta l’ingegnere della conoscenza a integrare nuove regole e mantenerle nel tempo e infine un **user interface** che consente d'interagire col sistema esperto.

- 


### b) Quali sono i vantaggi/svantaggi dei sistemi esperti

- i problemi dei sistemi esperti sono diversi, in primis la modellazione della conoscenza non è efficiente il che lo rende inutilizzabile per domini troppo complessi. Parlando sempre di dominio, su alcuni domini già esistenti ci sono algoritmi tradizionali che performano meglio di un sistema esperto → per esempio, usando un sistema esperto si può risolvere un problema di pianificazione ma non disponiamo dei formalismi più adeguati e sopratutto ci sono algoritmi tradizionali che risolvono il problema in modo più efficiente.

## 38) Costruzione (Inference Engine, KB, WM, Facilities, UI)

- Abbiamo diverse componenti:
  - **Knowledge-Base** → (Rule Base), mantiene la conoscenza dell'esperto come regole *condizione-azione*
  - **Working Memory (WM)** → mantiene i fatti iniziali e quelli generati dalle inferenze
  - **Inference Engine**
    - *Pattern matching:* confronta la parte *if* delle regole rispetto ai fatti della WM
    - regole che hanno la parte *if* soddisfatta sono dette **attivabili** e sono poste nell'Agenda
    - **Agenda** → elenco ordinato di regole attivabili. Le regole sono ordinate in base alla loro priorita' o in base ad altre strategie di preferenza/conflcit resolution.
    - **Execution** → la regola in cima all'agenda e' selezionata ed eseguita (firing)
  - **Explanation Facility** → fornisce una giustificazione delle soluzioni (reasoning chain equivalente a sequenza di regole che sono state attivate)
  - **Knowledge Acquisition Facility** → aiuta l'ingegnere della conoscenza a integrare nuove regole e a mantenerle nel tempo
  - **User Interface** → consente all'utente d'interagire con il sistema esperto

## 39) Conoscenza di dominio / conoscenza di controllo

- Per **conoscenza di dominio** intendiamo che il nostro sistema esperto ha una conoscenza specifica del dominio di riferimento, questa è infatti una delle proprietà di un sistema esperto.
  Per **conoscenza di controllo** intendiamo che il nostro sistema esperto ha una conoscenza basata sul COME utilizzare la conoscenza a disposizione sul dominio per risolvere problemi

## 40) Quando non utilizzare un S.E.

- Non ha senso utilizzare un sistema esperto quando → esistono algoritmi ”tradizionali” efficienti per risolvere il problema considerato, l’aspetto principale del problema `e la computazione e non la conoscenza,  la conoscenza non può essere modellata o esplicitata in modo efficiente e infine quando l’utente finale `e riluttante ad applicare un sistema esperto per via della criticità del task

## 41) Meccanismi di un motore inferenziale

- Il motore inferenziale di CLIPS viene attivato tramite il comando *run*. Questo comando fa si che il motore inferenziale esegua una catena di attivazioni di regole fino a quando:
  - Il comando (halt) compare nel conseguente di una regola e quindi viene sospeso il motore di CLIPS.
  - Non ci sono più regole attivabili (agenda vuota).
  - Possiamo avere anche il comando *(run n)* dove *n* è il numero interno che indica il numero massimo di regole che si vogliono eseguire, poi il motore si ferma anche se l’agenda non è vuota.

  Vediamo quindi come funziona il motore inferenziale di CLIPS:
  - **Determina** tutte le regole attivabili dai fatti presenti in WM (fare pattern matching tra tutti i fatti in WM e le regole)
  - **Ordina** le regole attivabili in agenda in base al criterio di conflict resolution (il pattern matching ha lo scopo d'identificare le regole attivabili, ovvero regole che hanno precondizioni vere).
  - **Fire** della regola in cima all’agenda
  - Torno al punto 1.

# CLIPS

## 42) Come funziona CLIPS: qual è il suo ciclo d’esecuzione

- CLIPS e’ un sistema a regole di produzione il cui componente principale sono i fatti. Tali fatti possono essere:
  - Ordinati: conta l’ordine degli elementi all’interno del fatto stesso
  - Non Ordinati: l’ordine non conta.

  Il funzionamento di CLIPS e’ il seguente:
  - **Avvio di CLIPS** tramite riga di comando
  - **Definisco i fatti**
  - **Modello la conoscenza tramite i templates**
  - **Manipolo i fatti**

- **Produco delle regole**
- **Definisco eventuali variabili**
- **Avvio il motore inferenziale di CLIPS** tramite *run*
- **Eseguo le regole**
- **Termino**

- Per ciclo di esecuzione magari si intende il funzionamento del motore inferenziale:
    - **Determina** tutte le regole attivabili dai fatti presenti in WM (fare pattern matching tra tutti i fatti in WM e le regole)
    - **Ordina** le regole attivabili in agenda in base al criterio di conflict resolution (il pattern matching ha lo scopo d'identificare le regole attivabili, ovvero regole che hanno precondizioni vere).
    - **Fire** della regola in cima all'agenda
    - Torno al punto 1.

## 43) CLIPS di base: tipi fatti ordinati non ordinati, manipolazione.

- CLIPS e' un sistema a regole di produzione il cui componente principale sono i fatti. Tali fatti possono essere:
    - Ordinati : conta l'ordine degli elementi all'interno del fatto stesso
    - Non Ordinati: l'ordine non conta → possiamo avere un template

  In CLIPS possiamo **aggiungere un nuovo fatto** tramite l'istruzione *assert(x)* che aggiunge un nuovo fatto *x* all'interno della mia KB. Possiamo anche costruire un template per costruire fatti piu' complessi, attraverso il quale posso andare a manovrare fatti complessi con maggiore semplicita'. Il comando e' *deftemplate(x).* E' possibile anche dare dei fatti non necessariamente completi di tutti gli attributi, come ad esempio: (*assert (person (nome "John Brown"))).* Inoltre **possiamo andare a modificare un fatto gia' esistente,** ottenendo un nuovo fatto che verra' sovrascritto all'interno della mia KB. Oltre a modificarlo, **un fatto puo' essere duplicato** tramite *duplicate*, in cui vado a duplicare un determinato fatto, andando a modificare il valore di tutti gli attributi *x* a cui assegniamo *v*. Oltre a essere duplicati, i fatti possono anche essere **rimossi** dalla WM tramite il comanda (retract <fact>)
  I fatti sono effettivamente inseriti all'interno della WM dopo aver dato il comando (reset), mentre per elencare i fatti si usa (facts). Inoltre, **abbiamo la fase d'ispezione della WM** tramite (facts) e (wacth facts) in cui automaticamente vengono mostrati i cambiamenti che occorrono nella WM a seguito dell'esecuzione delle regole.

## 44) Fatti ordinati e non ordinati in CLIPS: come si definiscono che differenze ci sono

- CLIPS e' un sistema a regole di produzione il cui componente principale sono i fatti. Tali fatti possono essere:
    - Ordinati: conta l'ordine degli elementi all'interno del fatto stesso
    - Non Ordinati: l'ordine non conta →possiamo avere un template

## 45) Come è definita una regola in CLIPS: cosa è presente nella parte sinistra e cosa nella parte destra

- In CLIPS possiamo definire un insieme di regole tramite il comando *defrule*. Affinché una regola possa essere attivata, devono valere delle precondizioni. Sostanzialmente una regola e' formata da una parte sinistra (LHS) in cui e' presente l'antecedente della regola e una parte destra (RHS) in cui e' presente il conseguente della regola. Vediamolo con un esempio: *data una regola definita EmergencyRule, possiamo dire che come antecedente abbiamo una certa emergenza di tipo A, mentre il conseguente di una regola, sarebbe il fatto che in presenza di un emergenza scatta l'allarme.*

  In CLIPS inoltre abbiamo l'agenda, il quale mostra l'elenco ordinato di tutte le possibili regole attivabili, in cui la regola in cima sarà' la prossima ad attivarsi e, per ogni regola, sono indicati i fatti che lo attivano. Abbiamo poi il comando (watch rules) che mostra, durante l'esecuzione, quali regole vengono eseguite.

## 46) Cos'è la rifrazione in CLIPS e che scopo ha; cosa succede se i fatti vengono ritrattati e riasseriti?

- E' una situazione utilizzata dai sistemi esperti in CLIPS per evitare possibili situazioni di loop, impedendo quindi di attivare una stessa regola più volte utilizzando gli stessi fatti. E' un meccanismo alla base del pattern-matching che impedisce di attivare due volte una stessa regola sugli stessi fatti. *Ad es. La regola scatta una volta sola per ogni persona con gli occhi blu.* Se i fatti vengono ritrattati vengono eliminati dalla WM, mentre se vengono riasseriti, verranno inseriti nuovamente all'interno della WM con i nuovi fatti associati ad essi.

## 47) Qual è la strategia principale che segue CLIPS per ordinare le regole in agenda?

- Nell'agenda, la regola al top e' la prima ad essere eseguita. Tuttavia per capire come ordinare le regole, e' possibile determinarle come segue:
  - **Salience** (priorità') → tutte le regole con salience più bassa sono sotto una certa soglia corrente, mentre quelle con salience più alta sono sopra una certa soglia e verranno eseguite nell'ordine
  - A parità di salience, viene determinato dalla **strategia di risoluzione del conflitto** a determinare la posizione.
  - Se una regola viene attivata dagli stessi fatti della WM, e i passi precedenti non sono in grado di stabilire un ordine, allora l'ordine con cui le regole sono scritte stabiliscono la priorità'.

## 48) Perché cambiando l'ordine delle regole in agenda il risultato finale del programma potrebbe essere diverso?

- Perché potrebbero andare in cima allo stack altre regole e quindi essere eseguite prima di altre, producendo un risultato diverso. (Questo però solo a parità di salience e di strategie di conflitto)

## 49) Regole (LHS, RHS, Binding)

- LHS, RHS dette sopra. Binding:

## 50) Cos'è: l'agenda? (luglio2022) o

- Come funziona l'ordinamento nell'agenda
- Se abbiamo in agenda R1 e sotto R2 e eseguiamo R1, R2 rimane in agenda?

- L'agenda e' situato all'interno del motore inferenziale di CLIPS. Rappresenta un elenco ordinato di regole attivabili (lista di tutte le regole che hanno la parte sinistra LHS soddisfatta e non sono state ancora eseguite), le cui regole sono ordinate in base alla loro priorita', strategia di preferenza/conflict resolution. Nell'agenda, la regola al top e' la prima a essere eseguita. Tuttavia per capire come ordinare le regole, e' possibile determinarle come segue:
  - **Salience** (priorita') → tutte le regole con salience piu bassa sono sotto una certa soglia corrente, mentre quelle con salience più alta sono sopra una certa soglia e verranno eseguite nell'ordine
  - A parità di salience, viene determinato dalla **strategia di risoluzione del conflitto** a determinare la posizione.
  - Se una regola viene attivata dagli stessi fatti della WM, e i passi precedenti non sono in grado di stabilire un ordine, allora l'ordine con cui le regole sono scritte stabiliscono la priorita'.

Notiamo inoltre che se abbiamo in agenda due regole R1 ed R2, ed R1 viene eseguita prima di R2, successivamente la regola R2 continua a rimanere in agenda, perche' e' alla regola R1 che dopo essere eseguita viene fatto il pop, mentre R2 continua a rimanere.

## 51) Quali sono le strategie di conflict resolution che sono predefinite in CLIPS

- vediamo quali sono le possibili strategie di risoluzione dei conflitti:
  - **depth**: ha la precedenza la regole attivata dal fatto più recente
  - **breadth**: ha la precedenza la regola attivata da fatto più vecchio
  - **simplicity**: ha la precedenza la regola che effettua meno confronti espliciti al suo interno
  - **complexity**: ha la precedenza la regola che effettua piu confronti espliciti al suo interno
  - **lex**: si basa su ordinamento lessicografico
  - **mea:** variante del lex in cui i fatti non vengono riordinati, e dopo si stabilisce un ordinamento lessicografico.
  - **random:** viene scelta una regola casualmente

## 52) Che ruolo giocano gli indici dei fatti nelle varie strategie?

- Per la strategia **Depth** abbiamo che:
  - Dati due fatti *f1 → fatto-a attiva la regola-1 e regola-2*

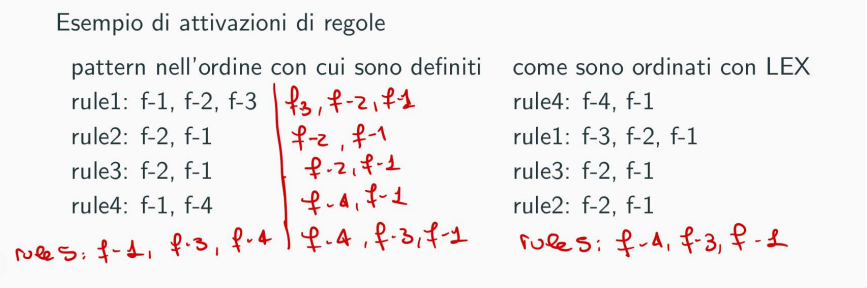
*f2 → fatto-b attiva regola-3 e regola-4*

- *Se viene asserito il fatto-a prima del fatto-b, allora la regola-3 e regola-4 saranno poste al di sopra della regola-1 e della regola-2 nell'agenda; la posizione della regola-1 relativa alla regola-2 e della regola-3 relativa alla regola-4 dipendera' dall'ordine con cui sono scritte le regole nel file sorgente.*

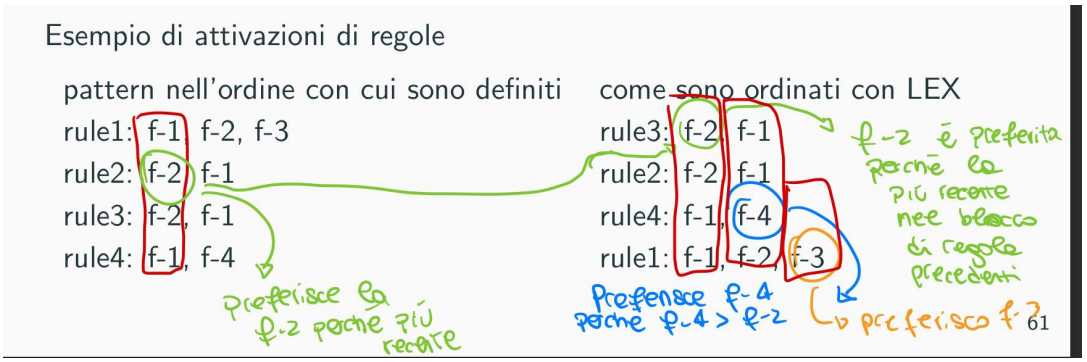
- Per la strategia **Breadth** abbiamo che:
  - f1 → *fatto-a attiva la regola-1 e la regola-2*
    *f2 → fatto-b attiva la regola-3 e regola-4*

*Se viene asserito il fatto-a prima del fatto-b, allora la regola-1 e la regola-2 saranno poste al di sopra della regola-3 e della regola-4 nell'agenda; la posizione della regola-1 relativa alla regola-2 e della regola-3 relativa alla regola-4 dipendera' dall'ordine con cui sono scritte le regole nel file sorgente.*

- Per la strategia **LEX** abbiamo che:
  - Ogni fatto e' etichettato con un "time tag" per indicarne la recentezza rispetto agli altri fatti
  - i pattern di ogni attivazione sono associati al time tag del fatto che li soddisfa
  - Un'attivazione con un pattern più recente e sistemato prima di un'attivazione con pattern meno recenti.




- Per la strategia **MAE** abbiamo che:
  - Ogni fatto e' etichettato con un "time tag" per indicarne la recentezza rispetto agli altri fatti
  - il primo time tag del pattern associato con il primo pattern e' utilizzato per determinare la posizione della regola
  - Un attivazione, in cui il primo time tag > altri primi tag di attivazione, e' inserita prima delle altre attivazione nell'agenda
  - A parita' di time tag del prima pattern entity, si confrontano i time tag dei pattern successivi.




• Come funzionano le clausole (forall ..) e (exists ..)

- **Forall:** (forall(pattern)) → soddisfatto se il pattern vale per tutti i fatti che unificano. Una regola quindi scatta solo se per tutti i valori di una certa condizione, e' anche stata asserita un altra condizione.
- **Exists:** (exists(pattern)) → soddisfatto per un unico fatto che unifica.

## 53) Moduli in CLIPS e come funzionano e a cosa servono **(domanda frequente)**

- CLIPS usa i moduli per partizionare la KB e la WM (e di conseguenza anche l'agenda). Ogni modulo ha la propria agenda e WM. In assenza d'indicazioni, tutti i fatti, template e regole sono definite nel modulo MAIN. I moduli devono poter scambiare qualche informazione, altrimenti averli partizionati non e' molto utile. Tutto quello che viene dichiarato all'interno di un modulo non e' visibile all'interno degli altri moduli. Quello che possiamo fare e' rendere "pubblici" certi elementi del nostro programma CLIPS tramite *import ed export*. Se esistono più moduli che invocano gli uni le regole degli altri, allora viene a formarsi uno stack di moduli che mi permette di stabilire il modulo in cui dobbiamo applicare una regola. Nel momento in cui non esiste più una regola applicabile per un certo modulo, tale modulo viene rimosso dallo stack e passiamo ad applicare regole successive. Quindi i moduli mi permettono di organizzare la mia KB e WM, in modo da poter capire (tramite principi di stack sui moduli) quali moduli e regole utilizzare per prima e organizzare computazionalmente il mio sistema esperto.

## 54) Regole e connettivi logici

- Sono i seguenti:
  - **and** → i pattern nell'antecedente delle regole sono considerati in congiunzione
  - **or** → soddisfatta se almeno uno dei due pattern unifica con i fatti della WM
  - **not** → soddisfatto quando nessun fatto unifica con il pattern
  - **exists** → soddisfatto per un unico fatto che unifica.
  - **forall** → soddisfatto se il pattern vale per tutti i fatti che unificano. Una regola quindi scatta solo se per tutti i valori di una certa condizione, e' anche stata asserita un altra condizione.

## 55) Ciclo di funzionamento di CLIPS

- 
- Seguente funzionamento:
    - Avviamento del motore inferenziale tramite il comando *run* che fa si che il motore inferenziale esegua una catena di attivazioni di regole finche o compare il comando *halt* che sospende il motore inferenziale oppure non ci sono piu regole attivabili.
    - Le regole vengono ordinate sulla base delle regole attivabili in agenda in base al criterio di conflict resolution (o salience delle stesse)
    - Faccio la fire ed eseguo le regole
    - Termino.

## 56) Field constraint

- Vincoli definiti per il singolo campo, usati per filtrare il pattern matching. Abbiamo:
    - **& and** → il valore del campo deve soddisfare una congiunzione di vincoli
    - **~not** → il campo puo' prendere qualsiasi valore tranne quello specificato
    - **| or** → sono specificati valori alternativi ammissibili per uno stesso campo
    - **: expression** → aggiungi l'espressione che segue come vincolo

N.B. In alternativa ai Field Constraint e' possibili usare la clausola test, che prende una funzione booleana e ne ritardo l'utilizzo dei vincoli.

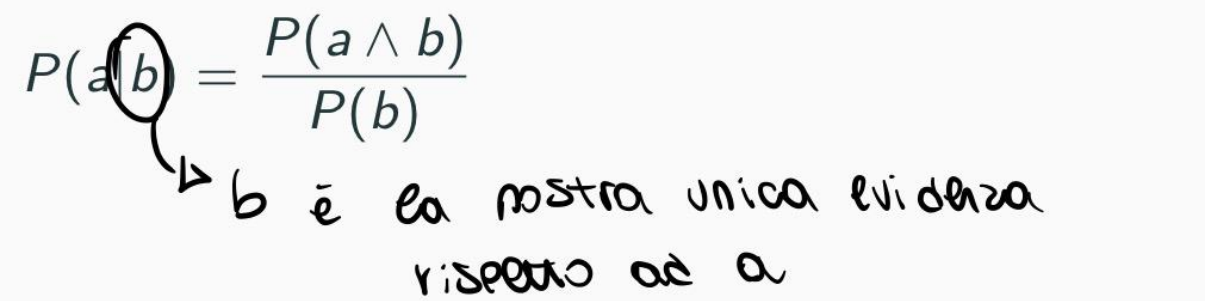
# Probabilità

## 57) Distribuzione di probabilità congiunta e indipendenza condizionale (luglio2022)

- Due variabili possono essere indipendenti in due modi: *indipendenti* e *condizionalmente indipendenti*.
    - **Indipendenti** → de variabili A e B sono dette indipendenti se vale che P(A|B) = P(A) oppure se P(A|B) = P(B), oppure se P(A v B) = P(A)P(B)
    - **Condizionalmente indipendenti** → consideriamo 3 variabili A,B,C diremo che A e' condizionalmente indipendente da B dato C, se valgono le seguenti condizioni:
        - P(A|B,C) = P(A|C)
        - P(B|A,C) = P(B|C)

      In generale l'indipendenza ci permettere di ridurre il numero di variabili in gioco, ma l'indipendenza condizionata e' lo strumento migliore che possiamo usare in ambienti in cui non abbiamo una totale conoscenza.

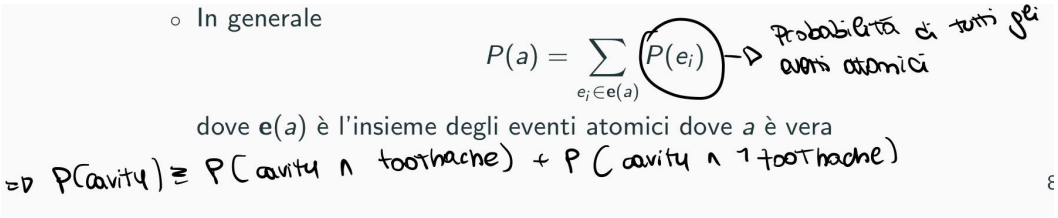
Possiamo inoltre definire la **distribuzione di probabilita' congiunta** come una matrice che assuma una probabilita' ad ogni possibile combinazione di valori della variabili considerate.




Abbiamo anche la **distribuzione di probabilità congiunta COMPLETA** se costruita su tutte le variabili del dominio.

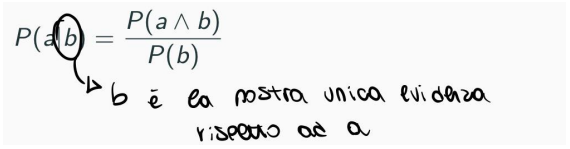
## 58) Evento atomico? (luglio2022)

- Un evento Atomico e' un evento che dà una specifica completa dello stato del mondo. Consiste in un assegnamento di valori a tutte le variabili casuali. Le proprietà sono le seguenti:
  - sono **mutuamente esclusivi**
  - l'insieme di tutti gli eventi atomici descrive la verità' o falsità' di qualsiasi proposizione (semplice o complessa che sia).
  - Ogni proposizione e' logicamente equivalente alla disgiunzione di tutti gli eventi atomici che ne implicano la verita'




## 59) Prob a priori/condizionata (luglio2022)

- **Probabilità a priori**→ e' il grado di credenza in una proposizione *a* in assenza di ogni altra informazione.
  Ad es. P(a); P(cavity) = 0.1; P(Weather = sunny) = 0.72
  **Probabilita condizionata** → P(a|b) e' la probabilità di *a* quando tutto ciò' che sappiamo e' *b*.




## 60) Probabilità condizionata dal punto di vista insiemistico

-

## 61) Bayes e indipendenza condizionale

- Per stabilire una probabilità, nei casi reali abbiamo a disposizione informazioni probabilistiche nella forma P(effetto|causa) e spesso vogliamo proprio calcolare quanto probabile sia una causa avendo osservato certi effetti. Dal teorema di Bayes abbiamo infatti che:
    - P(effetti|causa)P(effetti)P(causa) → P(causa|effetti)

  In questa situazione, possiamo quindi introdurre il concetto d'indipendenza **condizionale**, in cui consideriamo 3 variabili A,B,C e diremo che A e' condizionamene indipendente da B dato C, se valgono le seguenti condizioni:
    - P(A|B,C) = P(A|C)
    - P(A|B,C) = P(A|-C)

  Pertanto l'indipendenza condizionale e' lo strumento migliore quando vogliamo calcolare una certa probabilita' desiderata in mancanza di abbastanza conoscenza.

## 62) Classificatore bayesiano naive

- Il classificatore Bayesiano (definito anche Modello Ingenuo), può essere generalizzato come:
    - P(Causa, Effetto1,..., Effetto*N)* = P(Causa) ∏ P(Effetto *i* | Causa).
      In questo casa l'ingenuità' sta nel fatto che si assumo che gli *n* effetti siano tra loro indipendenti, mentre nei casi reali non sempre questo e' ragionevole.

## 63) Reti bayesiane e esercizio tabella cpt (luglio2022)

- Le reti bayesiano sono una rappresentazione grafica di asserzioni d'indipendenze condizionali. Una rete bayesiana e' un **grafo** in cui:
    - un nodo per ogni variabile casuale (discreta o continua)
    - e' un DAG in cui abbiamo archi diretti e privi di cicli
    - un arco X a Y implica che X e' un genitore di Y e X ha una **influenza diretta** su Y.
    - Ogni nodo Xi e' associata a una distribuzione condizionale di Xj dati i suoi genitori: *P(Xi | Parents(Xi))*.
      Quantifica gli effetti dei genitori sul nodo; e' espressa come *Tabella di Probabilita' Condizionali **(CPT)***.
    - In generale la CPT di una variabile boolean, con *k* genitori booleani contiene 2^*k* righe: una per ogni possibile combinazione di valori dei genitori. Sostanzialmente CPT rappresenta con che distribuzione di pribabilita' i nodi genitori influenzano il nodo stesso.
      Ogni riga contiene un valore *p* per Xi = True. Se ogni variabile non ha piu di *k* genitori, la rete completa richiede O(n * 2^k) valori. Dato che in generale k << n, il numero di valori necessari cresce linearmente in *n* contro O(2^n) della distribuzione completa congiunta.

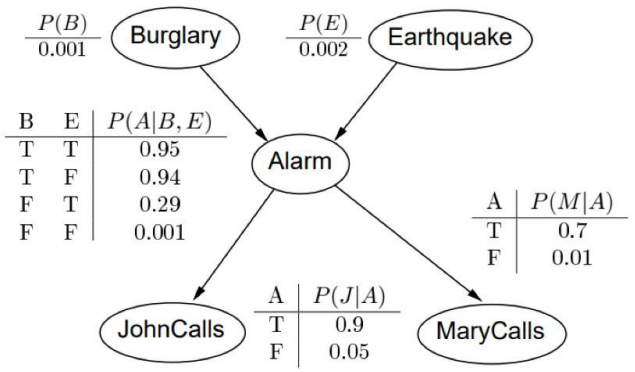



Figura 15: Rete bayesiana sul dominio del ladro

## 64) Semantica Globale delle BN

- La semantica globale definisce la distribuzione completa congiunta come il prodotto delle distribuzioni condizionali locali

$$\mathbf{P}(X_1, \ldots, X_n) = \prod_{i=1}^{n} \mathbf{P}(X_i | Parents(X_i))$$

Nelle reti bayesiane esiste anche il concetto di **semantica locale** in cui ogni nodo e' condizionalmente indipendente dai suoi non-discendenti dati i suoi genitori. Possiamo anche dire che un certo nodo e' condizionalmente indipendente da ogni altro nodo se siamo a conoscenza dei: genitori, figli e figli dei genitori. L'insieme di questi nodi viene detto **coperta di Markov.**

## 65) Costruire una BN

- Per costruire una rete dobbiamo fare attenzione all'ordine con cui i nodi sono aggiunti alla rete stessa. Sarebbe utilie un criterio per stabilire se un nodo deve essere un genitore di un altro nodo. Tuttavia e' possibile utilizzare la **chain rule** in cui:

$$\begin{aligned}
\mathbf{P}(X_1, \ldots, X_n) &= \mathbf{P}(X_n | X_{n-1}, \ldots, X_1)\mathbf{P}(X_{n-1}, \ldots, X_1) \\
&= \mathbf{P}(X_n | X_{n-1}, \ldots, X_1)\mathbf{P}(X_{n-1} | X_{n-2}, \ldots, X_1)\mathbf{P}(X_{n-2}, \ldots, X_1) \\
&= \ldots \\
&= \prod_{i=1}^{n} \mathbf{P}(X_i | X_{i-1}, \ldots, X_1)
\end{aligned}$$

Esiste una tecnica basata su alcuni semplici passi:

- Stabilire un ordinamento della variabili X1,..., X*n*
- Per ogni *i* da 1 a *n*, aggiungere X*i* alla rete selezionando un certo numero di genitori appartenenti a *X1,..., Xi-1* per cui vale che:

  *P(Xi | parents(Xi)) = P(Xi | X1,..., Xi-1)*

  In questo modo garantiamo la semantica globale in cui:

  *Parents(Xi) ⊂ {Xi-1,...., X1}*

# Glossario dei termini (probabilità):

- **Probabilità**: uno strumento per riassumere l'incertezza che deriva da pigrizia e ignoranza, assegnare una probabilità a una formula corrisponde al grado di credenza nella verità della formula.
    - **NB.** Probabilità dell'80% non vuol dire vera all'80%, non e un grado associato alla verità (come in fuzzy logic), ma alla credenza
- **Teoria delle decisioni** = Teoria delle Probabilità + Teoria dell'Utilità → non basta tenere i considerazione solo la probabilità di successo del piano ma anche l'utilità del piano (vedere esempio dell'aeroporto → se io parto un ora prima per andare i aeroporto sarò di sicuro li in tempo ma avrò un utilità bassissima.
- **Variabile casuale**: si riferisce a una parte del mondo il cui stato `e inizialmente sconosciuto; può avere dominio continuo o discreto → noi assegneremo a un variabile casuale un preciso valore.
    - Lettere minuscole a, b, c indicheremo variabili casuali generiche (sconosciute) (a, b, c)
    - lettere maiuscole indicheremo variabili casuali di un dominio specifico (Weather)
- **Evento atomico**: specifica completa dello stato del mondo ⇒ e un assegnamento di valori a tutte le variabili casuali, l'insieme di tutti gli eventi atomici descrive la verità o falsità di qualsiasi proposizione.
- **Probabilità a priori**: e il grado di credenza in una proposizione a in assenza di ogni altra informazione
- **Distribuzione di probabilità**: e un vettore di valori di probabilità → uno per ogni valore del dominio della variabile casuale considerata (P(Weather) = <0.7, 0.02, 0.08, 0.2>)
- **Distribuzione di probabilità congiunta**: e una matrice che assegna un valore di probabilità a ogni possibile combinazione di valori delle variabili considerate

| $Weather =$ | $sunny$ | $rain$ | $cloudy$ | $snow$ |
|---|---|---|---|---|
| $Cavity = true$ | | | | |
| $Cavity = false$ | | | | |

- **Distribuzione di probabilità congiunta completa**:una distribuzione di probabilità congiunta costruita su tutte le variabili del dominio.
- **Probabilità condizionata (posteriori)**: P(a|b) e la probabilità di a quando tutto ciò che sappiamo e b, la formula e la seguente → P(a|b) = P(a & b) * P(b).
- **Distribuzione di Probabilità Condizionate**: P(X, Y) = P(X|Y)P(Y) definita per ogni possibile coppia di valori X e Y, non e un prodotto tra matrici, ma tra gli elementi.
    - **Osservazione**: probabilità condizionate non sono "implicazioni logiche con incertezza" → P(a|b) = .8 non deve essere interpretata come "ogni volta che si verifica b allora a ha l'80% di probabilità di verificarsi (P(a) = .8)
- **Inferenza probabilistica**: calcolo delle probabilità a posteriori di proposizioni poste come interrogazioni (query) partendo dalle prove osservate.

$$P(a) = \sum_{e_i \in \mathbf{e}(a)} P(e_i)$$

| | *toothache* | | *¬toothache* | |
|---|---|---|---|---|
| | *catch* | *¬catch* | *catch* | *¬catch* |
| *cavity* | 0.108 | 0.012 | 0.072 | 0.008 |
| *¬cavity* | 0.016 | 0.064 | 0.144 | 0.576 |

$P(\text{cavity} \vee \text{toothache}) = 0.28$ (somma di quei 6 valori)

esempio di calcolo

$$P(\neg cavity | toothache) = \frac{0.016+0.064}{0.108+0.012+0.016+0.064} = 0.4$$

- . **Probabilità Marginale** ≡ Proprietà a priori

| | *toothache* | | *¬toothache* | | |
|---|---|---|---|---|---|
| | *catch* | *¬catch* | *catch* | *¬catch* | |
| *cavity* | 0.108 | 0.012 | 0.072 | 0.008 | 0.2 |
| *¬cavity* | 0.016 | 0.064 | 0.144 | 0.576 | |

$P(\text{cavity}) = 0.2 \Rightarrow$ **Probabilità Marginale** ≡ Proprietà a priori

- **Probabilità congiunta** = e la distribuzione di probabilità a priori delle variabili in Y
  - $$\mathbf{P}(\mathbf{Y}) = \sum_{\mathbf{z}} \mathbf{P}(\mathbf{Y}, \mathbf{z})$$
  - Y e Z sono insiemi di variabili casuali e z e un vettore di valori alle variabili in Z
- **Regola di condizionamento** → vediamo la probabilità congiunta come probabilità condizionata
  - $$\mathbf{P}(\mathbf{Y}) = \sum_{\mathbf{z}} \mathbf{P}(\mathbf{Y}|\mathbf{z})\mathbf{P}(\mathbf{z})$$

    - Nella maggior parte dei casi ci interessa calcolare la probabilità condizionata di qualche variabile, avendo a disposizione altre variabili come prova.
- **Indipendenza**: due variabili casuali sono indipendenti se P(a|b) = P(a) o P(b|a) = P(b) or P(a ∧ b) = P(a)P(b) → quando due variabili sono indipendenti questo ci permette di diminuire le dimensioni della distribuzione congiunta completa
- **Regola del prodotto**: P(a ∧ b) = P(a|b)P(b) = P(b|a)P(a) che ci porta alla regola di Bayes
- **Regola di Bayes:**
    - $$P(b|a) = \frac{P(a|b)P(b)}{P(a)}$$

      generalizzando:

      $$\mathbf{P}(Y|X) = \frac{\mathbf{P}(X|Y)\mathbf{P}(Y)}{\mathbf{P}(X)} = \alpha\mathbf{P}(X|Y)\mathbf{P}(Y)$$
- **Regola di Bayes e condizionamento all'evidenza**:
    - $$\mathbf{P}(Y|X,\mathbf{e}) = \frac{\mathbf{P}(X|Y,\mathbf{e})\mathbf{P}(Y|\mathbf{e})}{\mathbf{P}(X|\mathbf{e})}$$
- **Generalizzazione delle relazioni d'Indipendenza Condizionale:**
    - $$\mathbf{P}(X,Y|Z) = \mathbf{P}(X|Z)\mathbf{P}(Y|Z)$$

      • Se l'indipendenza condizionale vale, allora

      $$\mathbf{P}(X|Y,Z) = \mathbf{P}(X|Z)$$ → è ininfluente

      $\cancel{P(X|Y) = P(X)}$ !!!

      $$\mathbf{P}(Y|X,Z) = \mathbf{P}(Y|Z)$$
- .
- .
-