ANNO ACCADEMICO 2024/2025

Intelligenza Artificiale e Laboratorio

Teoria

Altair's Notes

UNIVERSITÀ DI TORINO

DIPARTIMENTO DI INFORMATICA

Premessa

Licenza

Formato utilizzato

Box di "Concetto sbagliato":

Concetto sbagliato 0.1: Testo del concetto sbagliato

Testo contente il concetto giusto.

Box di "Corollario":

Corollario 0.0.1 Nome del corollario

Testo del corollario. Per corollario si intende una definizione minore, legata a un'altra definizione.

Box di "Definizione":

Definizione 0.0.1: Nome delle definizione

Testo della definizione.

Box di "Domanda":

Domanda 0.1

Testo della domanda. Le domande sono spesso utilizzate per far riflettere sulle definizioni o sui concetti.

Box di "Esempio":

Esempio 0.0.1 (Nome dell'esempio)

Testo dell'esempio. Gli esempi sono tratti dalle slides del corso.

Box di "Note":

Note:-

Testo della nota. Le note sono spesso utilizzate per chiarire concetti o per dare informazioni aggiuntive.

Box di "Osservazioni":

Osservazioni 0.0.1

Testo delle osservazioni. Le osservazioni sono spesso utilizzate per chiarire concetti o per dare informazioni aggiuntive. A differenza delle note le osservazioni sono più specifiche.

1 Introduzione

1.1 Il Corso in Breve...

1.1.1 Motivazioni

Definizione 1.1.1: Intelligenza Artificiale

L'intelligenza artficiale (o IA, dalle iniziali delle due parole, in italiano) è una disciplina appartenente all'informatica che studia i fondamenti teorici, le metodologie e le tecniche che consentono la progettazione di sistemi hardware e sistemi di programmi software capaci di fornire all'elaboratore elettronico prestazioni che, a un osservatore comune, sembrerebbero essere di pertinenza esclusiva dell'intelligenza umana.

Note:-

Meh, in realtà l'IA è una disciplina di confine. Però le tematiche sono prettamente informatiche.

IA In breve:

- Area di ricerca dell'informatica.
- Si occupa di tutto ciò che serve per rendere un computer intelligente come un essere umano.
- Interessata a problemi *intelligenti*: problemi per cui non esiste/non è noto un algoritmo di risoluzione[1].

Note:-

Il cubo di Rubik non è un gioco intelligente >:(

Ci sono tante sotto-aree di ricerca:

- Rappresentazione della conoscenza e ragionamento.
- Interpretazione/sintesi del linguaggio naturale.
- Apprendimento automatico.
- Pianificazione.
- Robotica.

[1] Tris, il labirinto, etc.

Si collega a tante discipline, oltre all'informatica:

- Filosofia.
- Fisica.
- Psicologia.

Questo insegnamento ha l'obiettivo di approfondire le conoscenze di Intelligenza Artificiale con particolare riguardo alle capacità di un agente intelligente di fare *inferenze* sulla base di una *rappresentazione esplicita della conoscenza* sul dominio. In questo corso si faranno anche sperimentazione di metodi di ragionamento basati sul paradigma della *programmazione logica*, sull'uso di *formalismi a regole* (CLIPS) e su *reti bayesiane* (ragionamento probabilistico[2]).

Programma:

- Dal punto di vista metodologico saranno a rontate problematiche relative a:
  - Meccanismi di ragionamento per calcolo dei predicati del primo ordine.
  - Programmazione logica.
  - Ragionamento non monotono.
  - Answer set programming.
- Queste metodologie verranno a rontate dal punto di vista sperimentale con l'introduzione dei principali costrutti del *Prolog*, lo sviluppo di strategie di ricerca in Prolog e l'utilizzo dell'ambiente *CLINGO* nella risoluzione di problemi in cui sia necessaria l'applicazione di meccanismi di ragionamento non monotono e del paradigma dell'Answer Set Programming.

Domanda 1.1

E le novità dell'AI che vanno di moda?

Risposta: vengono trattate in altri corsi (TLN, RNDL, AAUT, ELIVA, AGINT).

[2] Odio la probabilità con tutto il mio cuore <3

2 Il PROLOG

Definizione 2.0.1: PROLOG

PROLOG (Programming Logic) è un *linguaggio dichiarativo* basato sul *paradigma logico*:

- Non si descrive cosa fare per risolvere un problema.
- Si descrive la situazione reale con *fatti* e *regole* e si chiede all'interprete di verificare se un *goal* segue oppure no secondo una logica classica.

Note:-

Il PROLOG è equivalente alla logica dei predicati del primordine.

2.1 Le Basi

Definizione 2.1.1: Fatti

Si rappresenta con dei *fatti* un dominio di interesse.

Esempio 2.1.1 (Fatto)

Fatto per descrivere che un alimento contiene più calorie di un altro:

- piuCalorico(wurstel, banana).
- Rappresenta il fatto che il würstel è un alimento maggiormente calorico rispetto alla banana.

Definizione 2.1.2: Regole

Si rappresentano le possibili inferenze con delle *regole*:

```
head := subgoal1, subgoal2, ..., subgoaln
```

Esempio 2.1.2 (Regola)

```
felino(X) := gatto(X)
```

Rappresenta la regola che permette di concludere che i gatti sono felini.

Idee di base del PROLOG:

- Regole ricorsive.
- L'interprete analizza i fatti e le regole nell'ordine in cui si trovano nel programma.
- Meccanismo di pattern matching per uni care variabili e termini.
- L'interprete, dato un programma, cerca di dimostrare un goal considerando fatti e applicando regole, nel secondo caso generando sotto-goal.

Definizione 2.1.3: Clausole

Le clausole sono i fatti o le regole. Contengono:

- Atomi:
  - Costanti.
  - Numeri.
- Variabili.
- Termini Composti, ottenuti applicando funtori a termini.

Note:-

Un programma PROLOG è un insieme di clausole.

Osservazioni 2.1.1

- L'estensione dei file PROLOG è 'pl'.
- In PROLOG le variabili hanno l'iniziale maiuscola.
- L'unica struttura dati nativa è la lista.
- Per eseguire swi: swipl.
- Per compilare: ['nomefile.pl'].
- Il comando ';' indica possibili alternative.
- Il comando 'trace.' consente un esecuzione passo per passo.
- '\+' rappresenta la negazione per fallimento.
- L'ordine è importante perché PROLOG "legge" dall'alto verso il basso.

Qualche predicato *built-in*:

- `var(X)`: indica se `X` è una variabile.
- `ground(X)`: indica se `X` è istanziata.
- `atom(X)`: indica se `X` è atomica.

2.1.1 Liste

Definizione 2.1.4: Lista

La *lista* è la struttura dati principale in PROLOG. Una lista è caratterizzata da una testa e da una coda:

- Testa: primo termine (a sinistra) della lista.
- Coda: la lista dei termini dal secondo (incluso) in poi.

Note:-

Rappresentata come [Head | Tail].

```
?- [1,2,3,4,5] = [Head | Tail].
Head = 1
Tail = [2,3,4,5] = [Head | Tail]
Yes
```

```
?- [a, ciao, [], 2, [1, saluti]] = [Head | Tail].
Head = a
Tail = [ciao, [], 2, [1, saluti]]
Yes
```

Figure 2.1: Le liste in PROLOG.

Predicati *built-in*:

- `length(Lista, N)`: ha successo se la `Lista` contiene `N` elementi.
- `member(Elemento,Lista)`: ha successo se la `Lista` contiene il termine `Elemento`.
- `select(Elemento,Lista,Rimanenti)`: rimuove `Elemento` da `Lista` e restituisce `Rimanenti`.

2.2 Interprete PROLOG

Domanda 2.1

Come avviene l'esecuzione di programmi PROLOG?

- Esecuzione mediante *backward chaining* in profondità.
- Si parte dal *goal* che si vuole derivare:
  - *Goal* = congiunzione di formule atomiche $G_1, G_2, \ldots, G_n$.
  - Si vuole dimostrare, mediante risoluzione, che il goal segua logicamente dal programma.
- Una regola $A :- B_1, B_2, \ldots, B_m$ è applicabile a G_i se:
  - Le variabili vengono rinominate.
  - A e G_i unificano.

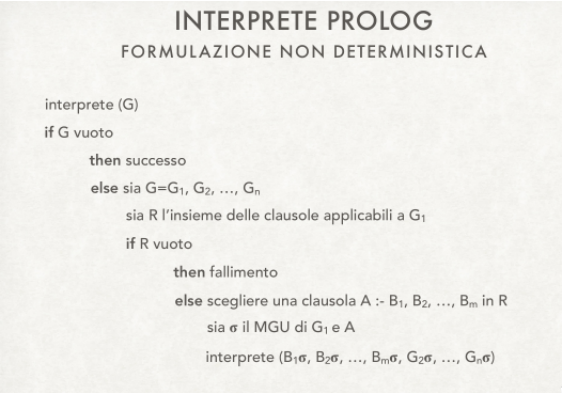


Figure 2.2: Una formulazione non deterministica di come funziona l'interprete PROLOG.

Note:-

MGU è il Most General Unifier: minimo sforzo per rendere uguali due variabili (il fatto e il goal).

- La computazione ha successo se esiste una computazione che termina con successo.
- Non determinismo: non è specificata la regola scelta in R.
- Ma l'interprete PROLOG si comporta in modo *deterministico*:
  - Le clausole vengono considerate nell'ordine in cui sono scritte nel programma.
  - Viene fatto backtracking all'ultimo punto di scelta ogni volta che la computazione fallisce.
- In caso di successo, l'interprete restituisce una sostituzione per le variabili che compaiono nel goal.

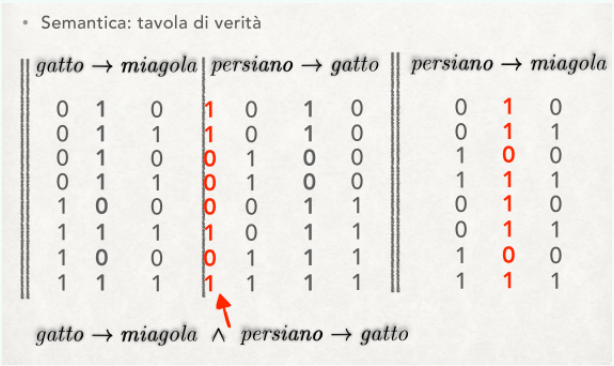
2.2.1 Breve Ripasso di Logica

Definizione 2.2.1: Logica Classica

Conseguenza logica definita semanticamente: dato una teoria e una formula, diciamo che la formula segue dalla teoria se essa è vera in tutti i modelli della teoria.

Esempio 2.2.1 (Gatti)

- I gatti miagolano: gatto $\longrightarrow$ miagola.
- I persiani sono gatti: persiano $\longrightarrow$ gatto.
- Si vuole dimostrare che i persiani miagolano: k $\vDash$ persiano $\longrightarrow$ miagola.

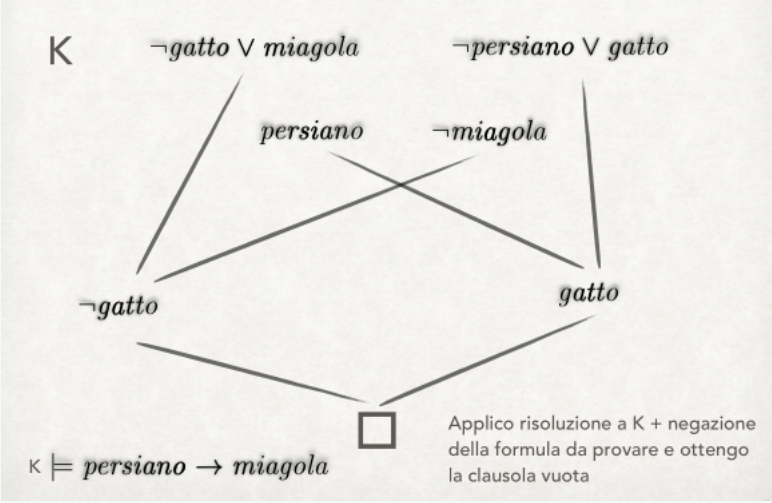


- Tuttavia il processo è molto laborioso già con poche formule e basi di conoscenza piccole.
- Metodo di prova: procedura/algoritmo che calcola/dimostra se una formula è conseguenza logica della teoria.
  - *Corretto*: se l'algoritmo dimostra F da K, allora F è conseguenza logica di K.
  - *Completo*: se F è conseguenza logica di K, allora l'algoritmo dimostra F da K.

Risoluzione:

- Si applica a formule in forma di *clausole* (disgiunzioni di letterali[1]).
- Si basa su un'unica regola di inferenza:
  - Date due clausole $C_1 = A_1 \vee \cdots \vee A_n$ e $C_2 = B_1 \vee \cdots \vee B_m$.
  - Se ci sono due letterali A_i e B_j tali che $A_i = \neg B_j$, allora posso derivare la clausola *risolvente* $A_1 \vee \ldots A_{i-1} \vee A_{i+1} \vee \cdots \vee A_n \vee B_1 \vee \ldots B_{j-1} \vee B_{j+1} \vee \cdots \vee B_m$.
  - Il risolvente è conseguenza logica di $C_1 \cup C_2$
- Data una teoria (insieme di formule) K e una formula F, dimostro che F è conseguenza logica di K per refutazione (dimostrare che $K \cup \neg F$ è inconsistente).
- Si parte dalle clausole $K \cup \neg F$, risolvendo a ogni passo due clausole e aggiungendo il risolvente all'insieme di clausole.
- Si conclude quando si ottiene la clausola vuota.

Esempio 2.2.2 (Risoluzione gatti)



Inoltre:

- Se le due clausole $C_1 = A_1 \vee \cdots \vee A_n$ e $C_2 = B_1 \vee \cdots \vee B_m$ contengono variabili, i due letterali A_i e B_j devono essere tali che si possa fare l'*unificazione* tra i due:
  - Unificazione: sostituzione α di variabili con termini o uguaglianza di variabili affinché $A_i = \neg B_j$.
  - Clausola risolvente $[A_1 \vee \ldots A_{i-1} \vee A_{i+1} \vee \cdots \vee A_n \vee B_1 \vee \ldots B_{j-1} \vee B_{j+1} \vee \cdots \vee B_m]\alpha$.
  - Le sostituzioni di α sono applicate a $A_1 \vee \ldots A_{i-1} \vee A_{i+1} \vee \cdots \vee A_n \vee B_1 \vee \ldots B_{j-1} \vee B_{j+1} \vee \cdots \vee B_m$.

[1] Formule atomiche o negazione di formule atomiche.

	costante c_2	variabile x_2	composto s_2
costante c_1	unificano se $c_1 = c_2$	unificano con x_2/c_1	non unificano
variabile x_1	unificano con x_1/c_2	unificano con x_1/x_2	unificano con x_1/s_2
composto s_1	non unificano	unificano con x_2/s_1	unificano se il funtore in s_1 e s_2 è lo stesso e gli argomenti unificano

Figure 2.3: Unificazione di due termini.

Note:-

Per ragioni d'efficienza, PROLOG non fa *occur check*, ossia una variabile X unifica con f(X).

2.2.2 Risoluzione SLD

Per arrivare a un linguaggio di programmazione PROLOG si vuole una strategia efficiente.

Definizione 2.2.2: Risoluzione SLD

Linear resolution with Selection function for Definite clauses:

- K con clausole *definite*:
  - Clausole di Horn: al più un letterale non negato.
  - Strategia linear input: a ogni passo di risoluzione, una *variante* di una clausola è sempre scelta nella K di partenza (programma) mentre l'altra è sempre il risolvente del passo precedente (goal, la negazione di F al primo passo).
  - Variante: clausola con variabili rinominate.

Note:-

{***NON*** LSD.

Domanda 2.2

Ma perché ci si limita alle clausole di Horn?

Risposta: si rimuove la parte "intuitiva" che non può essere implementata nel PROLOG. Inoltre le clausole di Horn garantiscono la completezza.

Derivazione SLD per un goal G_0 da un insieme di clausole K è:

- Una sequenza di clausole goal $G_0, G_1, \ldots, G_n$.
- Una sequenza di varianti di clausole di $KC_1, C_2, \ldots, C_n$.
- Una sequenza di MGU $\alpha_1, \alpha_2, \ldots, \alpha_n$, tali che G_{i+1} è derivato da G_i e da C_{i+1} attraverso la sostituzione α_{i+1},

Tre possibili tipi di derivazioni:

- Successo se G_n è vuoto (`true`).
- Fallimento finito, se non è possibile derivare da G_n alcun risolvente e G_n non è vuoto (`false`).
- Fallimento infinito, se è sempre possibile derivare nuovi risolventi (loop infinito).

Due forme di non determinismo:

- Regola di calcolo per selezionare a ogni passo l'atomo B_i del goal da unificare con una clausola.
- Scelta di quale clausola utilizzare a ogni passo di risoluzione.

Definizione 2.2.3: Regola di calcolo

Funzione che ha come dominio l'insieme dei goal e per ogni goal seleziona un suo atomo.

Note:-

La regola di calcolo non influenza correttezza e completezza del metodo di prova.

Domanda 2.3

Come si costruisce l'albero SLD?

Data una regola di calcolo, è possibile rappresentare tutte le derivazioni con un albero SLD:

- Nodo: goal.
- Radice: goal iniziale G_0.
- Ogni nodo $\leftarrow A_1, \ldots, A_m, \ldots, A_k$, dove A_m è l'atomo selezionato dalla regola di calcolo, ha un figlio per ogni clausola A $\leftarrow B_1, \ldots, B_k$ tale che A e A_m sono unificabili con MGU α. Il nodo figlio è etichettato con il goal $\leftarrow [A_1, \ldots, A_{m-1}, B_1, \ldots, B_k, A_{m+1}, \ldots, A_k]\alpha$. Il ramo dal padre al figlio è etichettato con α e con la clausola selezionata.

Scelte per rendere la strategia deterministica:

- Regola di computazione: *leftmost* (viene sempre scelto il sottogoal più a sinistra).
- Clausole considerate nell'*ordine in cui sono scritte nel programma*.
- Strategia di ricerca: *in profondità con backtracking*.
  - Non è completa perché se una computazione che porta al successo si trova a destra di un ramo infinito l'interprete non la trova, perché entra, senza mai uscirne, nel ramo infinito.

Note:-

Cercare di mettere a destra le computazioni che possano produrre eventuali casini.

2.2.3 Il Cut

Definizione 2.2.4: Cut

Il *cut* è un predicato extra-logico che consente di modificare l'esecuzione dell'interprete PROLOG. CUT (!):

- Predicato sempre vero.
- Se eseguito blocca il backtracking.

Note:-

Si rischia di perdere la completezza, ma si guadagna molto in efficienza.

Modello run-time dell'interprete PROLOG:

- Due stack:
  - Stack di *esecuzione*: contiene i record di attivazione (environment) dei predicati in esecuzione.
  - Stack di *backtracking*: contiene l'insieme dei punti di scelta (choice-point).
- In realtà c'è un solo stack, con alternanza di environment e choice-point.

Il cut:

- Rende definitive le scelte fatte nel corso della valutazione dall'interprete PROLOG (eliminazione di choice-point dallo stack di backtracking).
- Altera il controllo del programma.
- Perdità di dichiaratività.

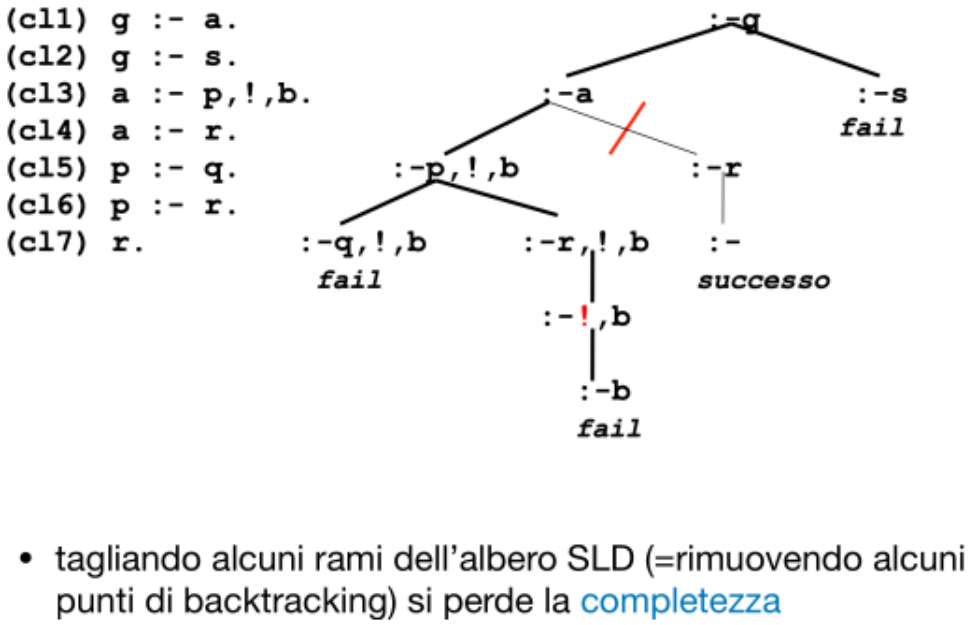


Figure 2.4: Esempio di cut che provoca la perdita di completezza.

2.3 Strategie di Ricerca in PROLOG

Un problema di ricerca è definito da:

- *Stato iniziale*.
- *Insieme delle azioni* (azione: fa passare da uno stato all'altro).
- Specifica degli obiettivi (goal).
- Costo di ogni azione.

Note:-
Non tutti i problemi hanno una naturale soluzione con la ricerca nello spazio degli stati.

2.3.1 Ricerca nello Spazio degli Stati

Lo spazio degli stati definito implicitamente dallo stato iniziale con un insieme delle azioni, ossia l'insieme di tutti gli stati raggiungibili a partire da quello iniziale.

Definizione 2.3.1: Cammino

Sequenza di stati collegati da una sequenza di azioni.

Corollario 2.3.1 Costo di un Cammino

Somma dei costi delle azioni che lo compongono.

Note:-
Se non si hanno dei costi espliciti si assume che siano tutti uguali (e. g. tutti 1).

Definizione 2.3.2: Soluzione a un Problema

Cammino dallo stato iniziale ad uno stato goal.

Corollario 2.3.2 Soluzione Ottima

Soluzione che ha il costo minimo tra tutte le soluzioni.

Note:-
Non è detto che esista una soluzione. In generale possono esistere 0, 1 o più soluzioni.

Stati rappresentati come termini:

- Dipendono dal problema da rappresentare:
  - Mondo dei blocchi: on(a,b), clear(c), ecc.
  - Puzzle dell'8: lista ordinata [3, 1, v, 4, 7, 8, 5, 6, 2].

Azioni specificate tramite:

- Precondizioni: in quali stati un'azione può essere eseguita.
- Effetti.
- applicabile(AZ, S): l'azione AZ è eseguibile nello stato S.
- trasforma(AZ, S, NUOVO_S): se l'azione AZ è applicabile a S, lo stato NUOVO_S è il risultato dell'applicazione di AZ allo stato S.

2.3.2 Cammini (Labirinto)

Specifiche:

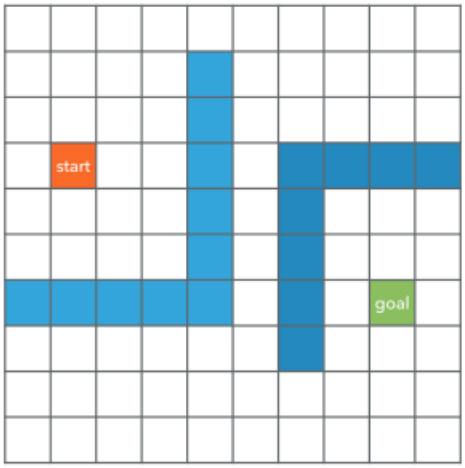
- Trovare un cammino in una griglia rettangolare, con ostacoli in alcune celle.
- Predicati num_righe e num_colonne speci cano la dimensione della griglia.
- pos(Riga,Colonna) per rappresentare la posizione dell'agente.
- occupata(pos(Riga,Colonna)) per rappresentategli ostacoli.

Azioni:

- Nord.
- Sud.
- Ovest.
- Est.

Azione applicabile quando la sua esecuzione non porta l'agente:

- Fuori dalla griglia.
- In una cella occupata da un ostacolo.




```
num_col(10).
num_righe(10).

occupata(pos(2,5)).
occupata(pos(3,5)).
occupata(pos(4,5)).
occupata(pos(5,5)).
occupata(pos(6,5)).
occupata(pos(7,5)).
occupata(pos(7,1)).
occupata(pos(7,2)).
occupata(pos(7,3)).
occupata(pos(7,4)).
occupata(pos(5,7)).
occupata(pos(6,7)).
occupata(pos(7,7)).
occupata(pos(8,7)).
occupata(pos(4,7)).
occupata(pos(4,8)).
occupata(pos(4,9)).
occupata(pos(4,10)).

iniziale(pos(4,2)).

finale(pos(7,9)).
```

Figure 2.5: Esempio di labirinto.

```
applicabile(nord,pos(R,C)) :-
    R>1,
    R1 is R-1,
    \+ occupata(pos(R1,C)).

applicabile(sud,pos(R,C)) :-
    num_righe(NR), R<NR,
    R1 is R+1,
    \+ occupata(pos(R1,C)).

applicabile(ovest,pos(R,C)) :-
    C>1,
    C1 is C-1,
    \+ occupata(pos(R,C1)).

applicabile(est,pos(R,C)) :-
    num_col(NC), C<NC,
    C1 is C+1,
    \+ occupata(pos(R,C1)).

trasforma(est,pos(R,C),pos(R,C1)) :- C1 is C+1.
trasforma(ovest,pos(R,C),pos(R,C1)) :- C1 is C-1.
trasforma(sud,pos(R,C),pos(R1,C)) :- R1 is R+1.
trasforma(nord,pos(R,C),pos(R1,C)) :- R1 is R-1.
```

Figure 2.6: Operazioni possibili.

Altri predicati extra-logici (asserzioni):

- `assert(Fatto(X))`: aggiunge fatti alla base di conoscenza.
- Può essere inserito in una regola.
- È un predicato *dinamico*.
- `asserta(Fatto(X))`: inserisce in testa (prima nell'ordine).
- `assertz(Fatto(X))`: inserisce in coda (dopo nell'ordine).
- `retract(Fatto(X))`: rimuove un fatto dalla base di conoscenza (ATTENZIONE: vale solo per i fatti inseriti dinamicamente da assert/asserta/assertz).
- `retractall(Fatto(_))`: rimuove tutti i predicati relativi al fatto.

2.3.3 Strategie di Ricerca

Definizione 2.3.3: Strategie non Informate

Strategie che non fanno assunzioni particolari sul dominio.

Strategie non informate:

- *Ricerca in profondità*:
  - Espande sempre per primo il nodo più distante dalla radice dell'albero di ricerca.
  - i può realizzare facilmente in Prolog sfruttando il nondeterminismo del linguaggio.
- *Ricerca a profondità limitata*:
  - Come per la ricerca in profondità, ma utilizzando un parametro che vincola la profondità massima oltre la quale i nodi non vengono espansi.
- *Iterative deepening*:
  - Ripete la ricerca a profondità limitata, incrementando a ogni passo il limite.
  - Ottima nel caso di azioni dal costo unitario.
- *Ricerca in ampiezza*:
  - Coda di nodi.
  - A ogni passo, la procedura espande il nodo in testa alla coda (`findall`) generando tutti i suoi successori, che vengono aggiunti in fondo alla coda.
  - Garantita l'individuazione della soluzione ottima.
- *Ricerca in ampiezza su grafi*:
  - Come la ricerca in ampiezza, ma considerando la lista chiusa dei nodi già espansi.
  - Prima di espandere un nodo, si veri ca che non sia chiuso.
  - Il nodo chiuso non viene ulteriormente espanso.

Definizione 2.3.4: Strategie Informate

Utilizzano una funzione euristica h(n)[a]. Si associa un costo a ciascuna azione e viene definita una funzione g(x)[b]

[a] Costo stimato del cammino più conveniente dal nodo n a uno stato finale.
[b] Costo del cammino trovato dal nodo iniziale a n.

Strategie Informate:

- *Ricerca in profondità IDA**:
  - Come iterative deepening, ma con soglia stimata a ogni passo in base all'euristica.
  - Al primo passo la soglia è h(si), dove si è lo stato iniziale.
  - A ogni iterazione, la soglia è il minimo f(n) = g(n) + h(n) per tutti i nodi n che superavano la soglia al passo precedente (backtracking).
  - si usa `assert` per salvare f(n) in caso di fallimento.
- *Ricerca in ampiezza con stima A**:
  - Ricerca in ampiezza su gra che tiene conto della funzione euristica.
  - A ogni passo si estrae per l'espansione dalla coda il nodo con minimo valore di f(n) = g(n) + h(n).
  - I nodi già espansi non vengono più espansi.

3 Answer Set Programming

3.1 Introduzione

Durante la cosiddetta "War of Semantics" nasce l'esigenza di dare una semantica alla negazione per fallimento adottata dagli interpreti PROLOG.

Definizione 3.1.1: Answer Set Programming

Paradigma di programmazione in cui le soluzioni sono i *modelli* (Answer Set), non più le prove

Note:-

Con PROLOG ha in comune solo la sintassi, per il resto è tutt'altra cosa.

L'Answer Set Programming (ASP):

- È particolarmente utile per risolvere problemi combinatori (soddisfacimento di vincoli, planning).
- ASP solvers molto efficienti sviluppati per supportare questa metodologia (DLV, smodels, *CLINGO*, Cmodels,...).

Note:-

Un ASP solvers è l'equivalente di un interprete PROLOG.

Codice ASP:

- Insieme finito di regole: $a :- b_1, b_2, \ldots, b_n, notc_1, notc_2, \ldots, notc_m.$
- a, b_i, c_j sono letterali nella forma p on $-p$:
  - - è la negazione classica.
  - not è la negazione per fallimento.
- a è opzionale, senza si ha *integrity constrain* (regole senza testa):
  - $:- a_1, a_2, \ldots, a_k.$
  - È inconsistente che siano tutti veri...
  - Serve per filtrare/buttare via dei modelli.
- Si applica ai soli programmi logici proposizionali.

- La maggior parte dei tool per ASP consente per comodità di usare variabili, ma le clausole devono poter essere trasformate in un numero finito di clausole ground.

ASP vs. PROLOG:

- In ASP l'ordine dei letterali non ha alcuna importanza.
- Prolog è goal-directed, ASP no.
- La SLD-risoluzione del Prolog può portare a loop, mentre gli ASP solver non lo consentono.
- PROLOG ha il cut(!), ASP no.

3.1.1 Negazione

- *Classica*:
  - attraversa :- treno.
  - Si attraversa solo se si può derivare che il treno non è in arrivo.
- *Per fallimento*:
  - attraversa :- not treno.
  - Si può attraversare in assenza di informazione esplicita sul treno in arrivo.
- Un letterale negato -p non ha nessuna proprietà particolare.
- Viene considerato come se fosse un nuovo atomo positivo, aggiungendo il vincolo :- p, -p.

Osservazioni 3.1.1

CLINGO:

- Fornisce modelli e indica se sono *SATISFIABLE* (soddisfacibili) o *UNSATISFIABLE* (insoddisfacibili).
- Se si aggiunge il parametro "0" vengono mostrati tutti i modelli (ATTENZIONE: potrebbero essere migliaia, è sconsigliato metterlo di default).

3.2 Semantica

Definizione 3.2.1: Answer Set

Un Answer Set è un modello minimale (stabile).

Note:-

Un programma ASP privo di letterali $notp_i$ ha un unico modello minimale che è il suo answer set. Potrebbero esserci più answer set, ma interessa solo quello minimale.

Domanda 3.1

Che succede se è presente la negazione per fallimento?

Definizione 3.2.2: Ridotto

Il *ridotto* P^S rispetto a un insieme di atomi S:

- Rimuove ogni regola il cui corpo contiene $notL$, per $L \in S$.
- Rimuove tutti i $notL$ dai corpi delle restanti regole.

P^S non contiene atomi con negazione per fallimento:

- Ha un unico answer set.
- Se tale answer ser coincide con S, allora S è un answer set per P.

Note:-

In ASP si può usare #show/cardinalità per mostrare solo alcune parti dei modelli.

Figure 3.1: "Show" or something idk.

4 Planning

Seconda parte del corso:

- Comprendere i problemi fondamentali alla base degli algoritmi di pianificazione.
- Studiare alcuni degli approcci classici alla pianificazione:
  - Assunzioni del planning classico.
  - Algoritmi di ricerca nello spazio degli stati (progression, regression, STRIPS).
  - Algoritmi di ricerca nello spazio dei piani (least-commitment planning).
  - Algoritmi di ricerca basati su grafi (grafo di pianificazione, GRAPHLAN).
  - Euristiche domain-independent per il planning.
  - Altri approcci al planning (HTN).
- Sistemi a regole:
  - Paradigma dei sistemi esperti basati su regole di produzione.
  - Sperimentare il paradigma a regole in CLIPS.
- Incertezza:
  - Modellare l'incertezza con le probabilità.
  - Meccanismi di inferenze probabilistiche.

4.1 Che Cos'è il Planning?

Definizione 4.1.1: Planning (secondo Haslum)

Il *planning* è l'arte e la pratica di pensare prima di agire.

Definizione 4.1.2: Automated Planning (Hoffman)

Selezionare un goal che motivi delle azioni basate su una descrizione ad alto livello del mondo.

In altre parole:

- Il planning è un processo deliberativo che sceglie ed organizza le azioni in base all'effetto che ci si aspetta queste producano.
- AI planning è lo studio della calcolabilità di questo processo deliberativo.

4.1.1 Modello Concettuale per il Planning

State Transition System $\Sigma = (S, A, E, \gamma)$

- Dove:
  - $S = \{s_1, s_2, \ldots\}$ insieme finito, ricorsivamente enumerabile di stati
  - $A = \{a_1, a_2, \ldots\}$ insieme finito, ricorsivamente enumerabile di azioni
  - $E = \{e_1, e_2, \ldots\}$ insieme finito, ricorsivamente enumerabile di eventi
  - $\gamma : S \times (A \cup E) \longrightarrow 2^S$ è una relazione di transizione di stato
- Se $a \in A$ e $\gamma(s, a) \neq \emptyset$, allora a è *applicabile* in s
- Applicare a in s causerà una transizione di stato del sistema da s a s', dove $s' \in \gamma(s, a)$

Osservazioni 4.1.1

Un STS $\Sigma = (S, A, E, \gamma)$ può essere rappresentato come un grafo diretto $G = (N_G, E_G)$ dove:

- $N_G = S$ è l'insieme dei nodi del grafo coincidente con l'insieme degli stati di Σ
- E_G è l'insieme degli archi del grafo tale che esiste un arco $s \xrightarrow{u} s'$ (anche rappresentato come $\langle s, u, s' \rangle$) da s a s' etichettato con $u \in A \cup E$, *se e solo se*:
  - $s, s' \in S$ e
  - $s' = \gamma(s, u)$

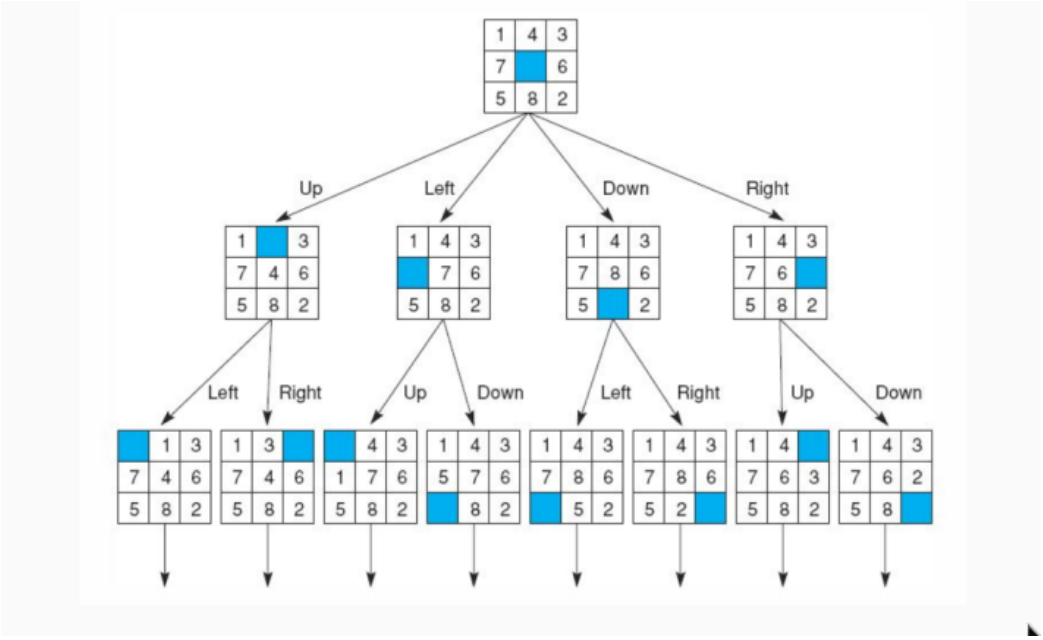


Figure 4.1: STS visto come grafo.

Ingredienti del planning:

- *STS*:
  - Descrive tutte le possibili evoluzioni del sistema.
- *Piano*:
  - Struttura che traccia le azioni necessarie per raggiungere un certo obiettivo G dato uno stato iniziale I.
  - È un cammino da I a G nello spazio degli stati tracciato da STS.
- *Goals*:
  - Un goal state s_g o un sottoinsieme di possibili goal state S_g.
  - Soddisfacimento di condizioni in tutta la sequenza di stati prodotta dalle azioni.
  - Ottimizzazione di funzioni di utilità.
  - Vincoli sulle azioni che possono essere eseguite.

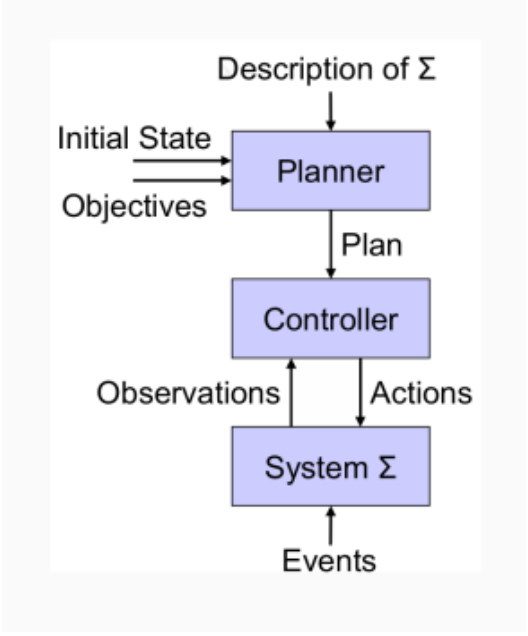


Figure 4.2: Si assume che gli eventi non interferiscano con il controller.

- *Planner:*
  - Data la descrizione di un STS Σ, lo stato iniziale, e il goal.
  - Genera un piano che raggiunge il goal dallo stato iniziale.
- *Controller:*
  - Dato un piano e lo stato corrente (funzione di osservabilità $\eta : S \rightarrow O$).
  - Seleziona ed esegue un'azione del piano.
- *STS Σ:*
  - Evolve in funzione delle azioni eseguite e degli eventi che possono accadere.

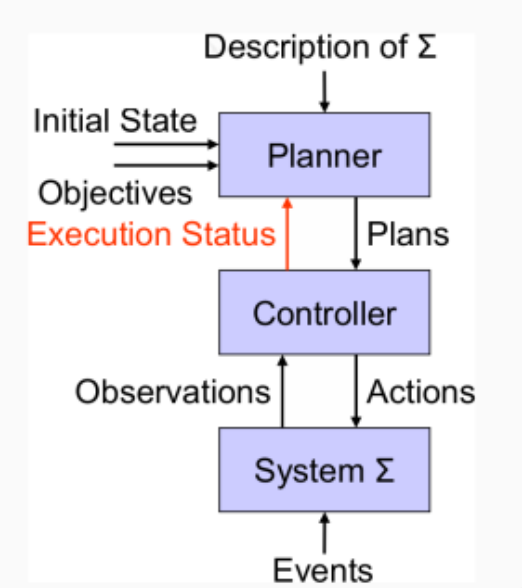


Figure 4.3: Gli eventi possono interferire.

- *Challenge:*
  - Il mondo reale può essere diverso da come è descritto nel modello.
- *Continual planning:*
  - Plan supervision.
  - Plan revision.
  - Re-planning.
- Continual planning consente un loop chiuso di feedback tra planner e controller.

4.1.2 Pianificazione Classica

Note:-

La pianificazione classica è la pianificazione che avviene sotto alcune assunzioni.

Definizione 4.1.3: Dominio Finito

Σ contiene un numero finito di stati.

Corollario 4.1.1 Rilassare un Dominio Finito (A0)

Per:

- Descrivere azioni che producono nuovi oggetti nel mondo.
- Trattare fluenti numerici.

Problemi:

- Decidibilità e terminazione del pianificatore.

Definizione 4.1.4: Dominio Completamente Osservabile

La funzione $\eta : S \rightarrow O$ è la funzione identità.

Corollario 4.1.2 Rilassare un Dominio Completamente Osservabile (A1)

Per

- Trattare state in cui non tutto è osservabile o può essere conosciuto.

Problemi:

- In genere si osserva solo un sottoinsieme della realtà, può accadere che $\eta(s) = \eta(s') = o$ con $s \neq s'$.
- Le osservazioni sono ambigue perché consistenti con più stati possibili.
- Determinare lo stato successore può essere problematico.

- Conformant planning (pianificazione in assenza di osservazioni): pianificare a prescindere dal reale stato del mondo.

Definizione 4.1.5: Dominio Deterministico

Σ è deterministico, cioè per ogni $s \in S, u \in A \cup E$ si ha $|\gamma(s,u)| \leqslant 1$.

Corollario 4.1.3 Rilassare un Dominio Deterministico (A2)

Per:

- Pianificare con azioni che possono avere risultati alternativi.

Problemi:

- Il controller deve osservare il risultato reale di ogni azione.
- Il piano soluzione potrebbe contenere dei branch condizionali o iterativi.

Definizione 4.1.6: Dominio Statico

Σ è statico, ovvero $E = \emptyset$ e STS può essere ridotto a $\Sigma = (S, A, \gamma)$.

Corollario 4.1.4 Rilassare un Dominio Statico (A3)

Per:

- Modellare domini in cui eventi al di là del controllo dell'esecutore sono possibili.

Problemi:

- Il mondo diventa non deterministico dal punto di vista del pianificare.

Definizione 4.1.7: Dominio con Goal Semplici

Consistono in uno stato s_g da raggiungere o un insieme di stati S_g (è sufficiente che il piano porti a uno di essi).

Note:-

Gli stati possono essere descrizioni parziali di situazioni desiderate.

Corollario 4.1.5 Rilassare un Dominio con Goal Semplici (A4)

Per:

- Trattare vincoli su stati e piani, funzioni di utilità/costo, ottimalità.

Problemi:

- Esprimere e ragionare su vincoli ulteriori nella specifica del goal rende il planning computazionalmente costoso.

Definizione 4.1.8: Dominio con Piani Sequenziali

Un piano soluzione è una sequenza finita di azioni linearmente ordinate.

Note:-

Una sola azione per volta è possibile.

Corollario 4.1.6 Rilassare un Dominio con Piani Sequenziali (A5)

Per:

- Sfruttare le capacità degli esecutori nel caso potessero eseguire più azioni.
- Non introdurre vincoli che non sono parte del dominio.

Problemi:

- Ragionare su e gestire strutture dati più complesse.

Definizione 4.1.9: Dominio con Tempo Implicito

Le azioni e gli eventi non hanno durata (oppure hanno durata istantanea).

Corollario 4.1.7 Rilassare un Dominio con Tempo Implicito (A6)

Per:

- Trattare azioni durative, problemi di concorrenza e deadline.

Problemi:

- Rappresentare e ragionare sul tempo.
- Gli effetti delle azioni si sviluppano nel tempo.

Definizione 4.1.10: Dominio con Single Agent

Un solo pianificatore e un solo controller (esecutore).

Corollario 4.1.8 Rilassare un Dominio con Single Agent (A7)

Per:

- Sfruttare meglio le risorse disponibili.
- Trattare situazioni in cui più esecutori sono presenti ma non sono sotto il controller di un unico pianificatore.

Problemi:

- Multi-agent planning: necessità di trattare le interazioni, coordinazione, competizione, negoziazione e planning della teoria dei giochi.

4.1.3 Problemi di Pianificazione Classica

Un problema di pianificazione classica è $P = (\Sigma, s_0, S_g)$:

- $\Sigma = (S, A, \gamma)$ è il modello del dominio espresso come STS.
- $s_0 \in S$ è lo stato iniziale.
- $S_g \subset S$ è l'insieme degli stati goal.

Una soluzione π a un problema P:

- Una sequenza totalmente ordinata di azioni istanziate (ground). $\pi = \langle a_1, a_2, \ldots, a_n \rangle$.
- Danno origine a una sequenza di transazioni di stato $\langle s_0, s_1, \ldots, a_n \rangle$ tale che:
  - $s_1 = \gamma(s_0, a_1)$.
  - $\forall k : 2..n s_k = \gamma(s_{k-1}, a_k)$.
  - $s_n \in S_g$.

Domanda 4.1

Quali sono le sfide dell'approccio classico al planning?

- Come rappresentare stati e azioni in modo da non dover esplicitamente enumerare S, A, e γ?
- Come ricercare una soluzione in modo efficiente? Quali algoritmi? Quali euristiche?
- Come generalizzare le soluzioni? Classical Planning troppo semplice per essere utile nei casi pratici, ma può essere la base per soluzioni in contesti più complessi (rilassando alcune assunzioni).

Domanda 4.2

Perché la pianificazione è difficile?

- È dimostrabile che il planning è un task computazionalmente costoso:
  - *PlanSAT*: esiste un piano che risolve un problema di pianificazione?
  - *Bounded PlanSAT*: esiste un piano di lunghezza k?
- Per la pianificazione classica entrambi i problemi sono decidibili (la ricerca avviene in spazio finito).
- Se si estende a uno spazio infinito:
  - PlanSAT diventa semi-decidibile: esiste un algoritmo che termina quando la soluzione non esiste, potrebbe non terminare quando la soluzione non esiste.
  - Bounded PlanSAT rimane decidibile.

Osservazioni 4.1.2

- Bounded PlanSAT è NP completo mentre PlanSAT è P.
- Trovare una soluzione è meno costoso che trovare una soluzione ottima.
- La complessità del planning giustifica la ricerca di euristiche, possibilmente domain-independent, che guidino il pianificatore nella sintesi di una soluzione.

Proprietà di un buon algoritmo di pianificazione:

- *Soundness* (correttezza): un pianificatore è corretto se tutte le soluzioni che trova sono piani corretti, ovvero realmente eseguibili dal controller:
  - Tutti i goals sono soddisfatti.
  - Nessuna precondizione di azione è open (mancante).
  - Nessun vincolo ulteriore è violato (vincoli temporali, istanziazione di variabili, etc.).
- *Completeness* (completezza):
  - Un pianificatore è completo se trova una soluzione quando il problema è risolubile.

  - Un pianificatore è strettamente completo se tutte le soluzioni sono mantenute nello spazio di ricerca (eventuali *pruning* dello spazio non scartano soluzioni).

- *Ottimalità*: un pianificatore è ottimo se l'ordine con cui le soluzioni sono trovate è coerente con una qualche misura di qualità dei piani (lunghezza, costo complessivo, etc.).

Note:-

Molti algoritmi, per favorire la velocità, rinunciano all'ottimalità.

4.2 Algoritmi di Pianificazione

4.2.1 Ricerca in Avanti e Ricerca all'Indietro

Definizione 4.2.1: Progression

Calcolo dello stato successore s' di uno stato s rispetto all'applicazione di un operatore o:

$$s' = \gamma(s, o)$$

Pianificatori basati su progression applicano delle ricerche in avanti tipicamente nello spazio degli stati:

- La ricerca comincia da uno stato iniziale.
- Iterativamente viene applicato un operatore o per generare un nuovo stato s'.
- La soluzione è trovata quando lo stato s' appena generato soddisfa il goal ($s' \vDash s_g$ e $s_g \in S_g$).
- Ha il vantaggio di essere molto intuitivo e facile da implementare.

```
function fwdSearch(O, s_0, s_g)
  state ← s_0
  π ← ⟨⟩
  loop
      if state.satisfies(s_g) then return π
      applicables ← {istanze ground da O applicabili in state}
      if applicables.isEmpty() then return failure
      action ← applicables.chooseOne()
      state ← γ(state, action)
      π ← π ∘ ⟨action⟩
  return failure
```

Figure 4.4: applicables.chooseOne() rappresenta una scelta non deterministica, ma esaustiva, di un'azione.

Note:-

chooseOne() avrà varie implementazioni a seconda dell'algoritmo scelto.

Corollario 4.2.1 Soundness di Progression

fwdSearch è corretto: se la funzione termina con un piano come soluzione, allora questo piano è effettivamente una soluzione al problema iniziale.

Corollario 4.2.2 Completeness di Progression

fwdSearch è completo: se esiste una soluzione allora esiste una traccia d'esecuzione che restituirà quella

soluzione come piano.

Osservazioni 4.2.1

- Il numero di azioni applicabili in un dato stato è in genere molto grande.
- Anche il branching factor tende a essere grande.
- La ricerca in avanti corre il rischio di non essere praticabile dopo pochi passi.

Ricerca in avanti vs. Ricerca all'indietro:

- La ricerca in avanti comincia da un singolo stato iniziale mentre la ricerca all'indietro comincia da un insieme di stati.
- Quando si applica in avanti un operatore o a uno stato s si genera un unico stato successore s', all'indietro possono esserci molteplici stati predecessori.
- Nella ricerca in avanti lo spazio di ricerca coincide con lo spazio degli stati, all'indietro ogni stato dello spazio di ricerca corrisponde a un insieme di stati del dominio.

Definizione 4.2.2: Regression

Il calcolo del regresso, ovvero il sottogoal predecessore di un goal dato, avviene nel seguente modo:

- Dato un goal g.
- Sia a azione ground tale che $g \in effects^+(a)$.
- $g' = \gamma^{-1}(g, a) = (g\ effects^+(a)) \cup pre(a)$.

g' è il regresso di g attraverso la relazione di transizione γ e l'azione a.

Nella ricerca all'indietro:

- Si comincia dall'insieme di stati goal.
- Iterativamente si seleziona un sottogoal generato precedentemente e si regredisce attraverso un operatore generando un nuovo sottogoal.
- La soluzione è trovata quando il nuovo sottogoal è soddisfatto dallo stato iniziale.
- Ha il vantaggio di poter gestire più stati contemporaneamente.
- È più costoso e difficile.

```
function bwdSearch(O, s0, g)
  subgoal ← g
  π ← ⟨⟩
  loop
      if s0.satisfies(subgoal) then return π
      relevants ← {ground instances from O relevant for subgoal}
      if relevants.isEmpty() then return failure
      action ← relevants.chooseOne()
      subgoal ← γ⁻¹(subgoal, action)
      π ← ⟨action⟩ ∘ π
```

Osservazioni 4.2.2

- La ricerca all'indietro si basa su azioni rilevanti, cioè quelle che contribuiscono attivamente al goal:
  - Producono almeno uno degli atomi che compaiono nel goal.
  - Non hanno effetti negativi sul goal stesso (non negano uno degli atomi del goal).
- La ricerca all'indietro mantiene un fattore di ramificazione più basso rispetto alla ricerca in avanti, ma il fatto di dover mantenere un belief state può complicare le strutture dati usate dal planner e la definizione di euristiche.
- Nonostante il vantaggio teorico la ricerca in avanti è preferita alla ricerca all'indietro.

4.2.2 STRIPS

Definizione 4.2.3: STRIPS

STanford Research Institute Problem Solver:

- Introduce una rappresentazione esplicita degli operatori di pianificazione.
- Fornisce una operalizzazione delle nozioni di:
  - Differenza tra stati.
  - Subgoal.
  - Applicazione di un operatore.
- Gestisce il Frame Problem.
- Si basa su due idee fondamentali:
  - Linear planning.
  - Means-End Analysis.

Osservazioni 4.2.3 Linear Planning

- Idea di base:
  - Risolvere un goal per volta, passare al successivo solo quando il precedente è stato raggiunto.
- L'algoritmo di planning mantiene uno *stack dei goal*:
  - Risolvere un goal può comportare la risoluzione di sottogoal.
- Conseguenze:
  - Non c'è interleaving nel conseguimento dei goal.
  - La ricerca è efficiente se i goal non interferiscono troppo tra loro.
  - Ma è soggetto alla *Sussmann's Anomaly*.

Osservazioni 4.2.4 Means-End Analysis

- Idea di base:
  - Considera solamente gli aspetti rilevanti al problema (backward).
  - Quali mezzi (means, operatori) sono disponibili e necessari per raggiungere il goal (end).

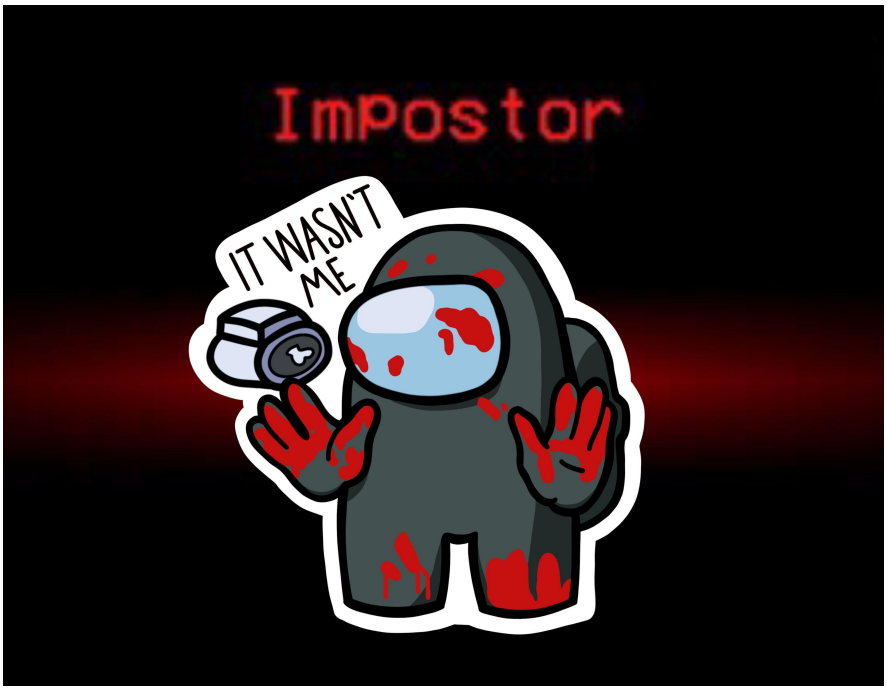


Figure 4.5: Sussmann's Anomaly caught in 4k.

- Occorre stabilire quali differenze ci sono tra lo stato corrente e il goal.
- Trovare un operatore che riduce tale differenza.
- Ripetere l'analisi sui sottogoal ottenuti per regressione attraverso l'operatore selezionato.

Caratteristiche di STRIPS:

- È un sottoinsieme della logica del primordine:
  - Numero finito di simboli di predicati, simboli di costanti e senza simboli di funzione né quantificatori.
- Uno stato in STRIPS è una congiunzione di *atomi ground* (privi di simboli di funzione).
- Vige l'*assunzione di mondo chiuso*: quello che non è descritto è considerato falso.
- Semantica degli insiemi:
  - Un atomo ground p vale in uno stato s se e solo se $p \in s$.
- Semantica logic-oriented:
  - Uno stato s soddisfa una congiunzione di letterali g, denotato come $s \vDash g$ se ogni letterale positivo in g occorre in s e se ogni letterale negativo in g non occorre in s.

Relazioni Fluenti:

- Predicati che rappresentano relazioni il cui valore di verità può cambiare da uno stato al successivo.

Relazioni Persistenti:

- Predicati il cui valore di verità non può cambiare.

Definizione 4.2.4: Plan Operators

Un operatore di pianificazione in STRIPS è una tripla: $o : (name(0), precond(o), effects(o))$ dove:

- $name(o)$ è una rappresentazione sintattica del tipo $n(x_1, \ldots, x_k)$ dove n è il nome dell'operatore e $x_1, \ldots, x_k$ è una lista di variabili che compaiono in o.
- $precond(o)$ è l'insieme di letterali che rappresentano le precondizioni dell'azione.
- $effect(o)$ è l'insieme di letterali che rappresentano gli effetti dell'azione.

Note:-

Una variabile può comparire negli effetti solo se è anche menzionata nelle precondizioni.

Definizione 4.2.5: Calcolo dello Stato Successore (progression)

Sia s uno stato del mondo ($s \in S$ per un dato dominio Σ). Sia a un'azione. Diremo che a è *applicabile* in s se e solo se:

- $precond^+(a) \subseteq s$.
- $precond^-(a) \cap s = \emptyset$.

La funzione di transizione di stato $\gamma(s, a)$ è definita:

- Se a è applicabile in s:
  - $\gamma(s, a) = (s\ effects^-(a)) \cup effects^+(a)$.
  - Altrimenti $\gamma(s, a)$ è indefinita.

Note:-

STRIPS usa la progression per modificare lo stato corrente a cui si giunge eseguendo le azioni fin qui selezionate nel piano in costruzione.

Definizione 4.2.6: Calcolo del Sottogoal Predecessore (regression)

Il calcolo del regresso avviene:

- Dato un goal g.
- Sia a azione ground tale che $g \in effects^+$(a).
- $g' = \gamma^{-1}(g, a) = (g\ effects^+(a)) \cup pre(a)$

Note:-

STRIPS usa la regression per ridurre la distanza tra il goal che si vuole raggiungere e lo stato corrente.

Il planner di STRIPS:

✓ Lo spazio di ricerca è ridotto perché i goal sono considerati uno per volta.

✓ Ideale quando i goal sono indipendenti tra loro.

✓ È sound.

✗ Può produrre piani subotttimi (Sussmann's Anomaly).

STRIPS (*initState*, *goals*)

`state` = *initState*; `plan` = []; `stack` = []

Push *goals* on `stack`

Repeat until `stack` is empty

- **If** top of `stack` is a goal g satisfied in *state*, **then pop** `stack`
- **Else if** top of `stack` is a conjunctive goal g, then
  Select an ordering for the subgoals of g, and **push** them on `stack`
- **Else if** top of `stack` is a simple goal sg, then
  Choose an operator o whose $effects^+$ matches goal sg
  Replace goal sg with operator o
  Push the preconditions of o on `stack`
- **Else if** top of `stack` is an operator o, then
  state = **apply**(o, *state*)
  `plan` = [`plan`; o]

✗ È incompleto (perché alcune azioni in alcuni domini non sono reversibili).

Definizione 4.2.7: Sussmann's Anomaly

È necessario disfare parte dei goal già raggiunti per poter risolvere l'intero problema.

Note:-

È una conseguenza di:

- Interdipendenza tra i sottogoal.
- Ordinamento sfavorevole dei goal.

Definizione 4.2.8: Planning Domain Definition Language

PDDL è uno standard per definire problemi e domini. Nella sua versione base è una sistematizzazione di STRIPS.

Estensioni di PDDL:

- *Conditional effects*: stabilire che alcuni effetti sono raggiunti solo se sono vere certe condizioni.
- *Durative actions*: un plan operator si divide in tre parti:
  - Un evento iniziale.
  - Un evento finale.
  - Una condizione invariante che deve permanere per tutta la durata dell'azione.
- *Numeric fluent*: consente di tracciare il consumo di risorse nel tempo.
- *Quantificators*.