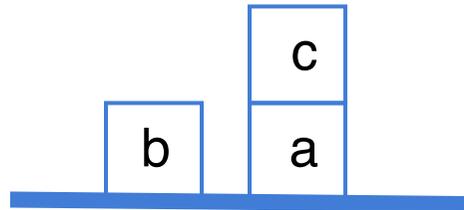

PLANNING LINEARE FORWARD

Esercizio in Prolog

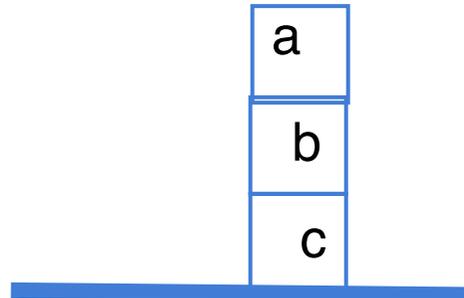
Planning lineare in Prolog

- Prendiamo l'esempio noto come anomalia di Sussman:

Stato iniziale:



Stato finale:



Stato

- Per rappresentare lo stato del problema, utilizzeremo un semplice linguaggio (di tipo logico):
 - $on_table(X)$ indica che il blocco X si trova sul tavolo;
 - $on(X, Y)$ indica che il blocco X si trova sopra il blocco Y
 - $clear(X)$ indica che il blocco X non ha nessun blocco al di sopra (ossia è la cima di una pila).
- Iniziale:
 $on_table(b) \wedge clear(b) \wedge on_table(a) \wedge on(c, a) \wedge clear(c)$
- Finale:
 $on_table(c) \wedge on(b, c) \wedge on(a, b) \wedge clear(a)$

Azioni

- ***move_to_table(X)***

precondizioni: *clear(X),on(X, Y)*

effetti: elimina *on(X, Y)*

aggiungi *on_table(X),clear(Y)*

- ***move_from_table(X,Y)***

precondizioni: *on_table(X),clear(X),clear(Y)*

effetti: elimina *on_table(X),clear(Y)*

aggiungi *on(X, Y)*

- ***move(X, Y)***

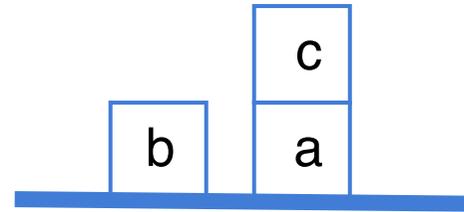
precondizioni: *clear(X),on(X,Z),clear(Y)*

effetti: elimina *clear(Y),on(X,Z)*

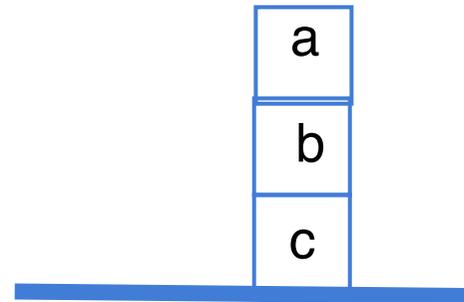
aggiungi *on(X, Y),clear(Z)*

Problema

Stato iniziale (I):



Stato finale (F):



- La sequenza di operatori dallo stato iniziale (I) allo stato finale (F) è:
 - *move_to_table(c)*,
 - *move_from_table(b,c)*,
 - *move_from_table(a,b)*

Planning come ricerca forward

- Si parte dal nodo iniziale e l'albero (o grafo) viene espanso applicando, ad ogni nodo, gli operatori le cui precondizioni sono soddisfatte nello stato che il nodo stesso rappresenta.
- La costruzione del grafo e la ricerca di uno stato finale può avvenire in modi differenti:
 - ricerca in **profondità**;
 - ricerca in **ampiezza**;
 - ricerca **euristica**.

Rappresentazione in Prolog

- Usiamo liste di *termini Prolog*:

%init

[on_table(b), clear(b), on_table(a), on(c,a),clear(c)]

%final

[on_table(c), on(b,c), on(a,b), clear(a)]

- Le precondizioni e le azioni degli operatori possono essere definite mediante due relazioni:

- **applicabile(OP, S)**

"l'operatore OP è applicabile nello stato S"

- **trasforma(OP, S, NUOVO_S)**

"NUOVO_S è lo stato risultante dall'applicazione dell'operatore OP allo stato S"

Azioni in Prolog - precondizioni

% 1)Precondizioni degli operatori

**applicabile(move_to_table(X), 1, S) :-
 member(clear(X), S),
 member(on(X,Y), S).**

**applicabile(move_from_table(X,Y), 1, S) :-
 member(on_table(X), S), member(clear(X), S),
 member(clear(Y), S), X\=Y.**

**applicabile(move(X,Y), 1, S) :-
 member(clear(X), S), member(on(X,Z), S),
 member(clear(Y), S), X\=Y.**

Azioni in Prolog - trasformazione

- La relazione *trasforma* permette di definire gli effetti delle azioni.
- Ogni trasformazione corrisponde all'eliminazione di un certo insieme di asserzioni ed alla aggiunta di altre asserzioni:

% 2) Definizione delle azioni

```
trasforma(move_to_table(X),S,[on_table(X), clear(Y) | S1]) :-  
    my_delete(on(X,Y),S,S1).
```

```
trasforma(move_from_table(X,Y),S, [on(X,Y) | S2]) :-  
    my_delete(on_table(X),S,S1),  
    my_delete(clear(Y),S1,S2).
```

```
trasforma(move(X,Y),S,[on(X,Y),clear(Z) | S2]) :-  
    my_delete(clear(Y),S,S1),  
    my_delete(on(X,Z),S1,S2).
```

Stato in Prolog

% 3)STATO INIZIALE E FINALE

iniziale([on_table(b),clear(b),on_table(a), on(c,a), clear(c)]).

finale(S) :-

member(on_table(c),S),

member(on(b,c), S),

member(on(a,b),S),

member(clear(a),S).

Predicati comuni

- Verificare l'uguaglianza tra due stati significa verificare che due liste contengano lo stesso insieme di asserzioni:

uguale([], []).

**uguale([X | REST1], S2) :-
member(X, S2),
my_delete(X, S2, REST2),
uguale(REST1, REST2).**

Predicati comuni

my_delete(EI, [], []).

**my_delete(EI, [EI|T], T1):-!,
my_delete(EI, T, T1).**

**my_delete(EI, [H|T], [H|T1]):-
my_delete(EI, T, T1).**

my_intersection([], S2, []).

**my_intersection([H|T], S2, [H|T3]):-
member(H, S2), !,
my_intersection(T, S2, T3).**

**my_intersection([H|T], S2, S3):-
my_intersection(T, S2, S3).**

Riassumendo:

Descrizione generale del problema:

- *applicabile(OP,S)*
"l'operatore OP è applicabile nello stato S"
- *trasforma(OP,S,NUOVO_S)*
"NUOVO_S è lo stato risultante dall'applicazione dell'operatore OP allo stato S"
- *uguale(S1,S2)*
"S1 e S2 rappresentano lo stesso stato"

Riassumendo:

Descrizione dell'istanza particolare da risolvere:

- *iniziale(S)*

"S è lo stato iniziale dell'istanza di problema che si vuole risolvere"

- *finale(S)*

"S è uno stato finale dell'istanza di problema che si vuole risolvere"

Ricerca depth-first

- Si parte dallo stato iniziale;
- se si raggiunge uno stato finale ci si ferma;
- in uno stato S non finale:
 - si seleziona un operatore OP applicabile in S ;
tale selezione definisce un **punto di scelta** nel processo di risoluzione
 - si applica OP determinando il nuovo stato $NUOVO_S$ da cui sarà proseguita la ricerca;
nel determinare tale stato si deve verificare che lo stato non sia già stato visitato (**lista dei nodi già visitati**).

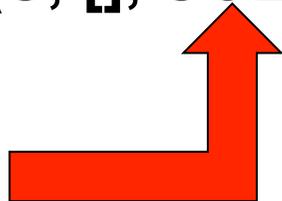
Ricerca depth-first

- **SOLUZIONE** è una sequenza di operatori che costituisce una soluzione del problema:

soluzione(SOLUZIONE):-

iniziale(S),

solve(S, [], SOLUZIONE).



lista degli stati già visitati

Ricerca depth-first

- **solve** è in grado di fornire, in backtracking, tutte le soluzioni per un problema:

```
solve(S, _, []) :-  
    finale(S), !.
```

```
solve(S, VISITATI, [OP | RESTO]):-  
    applicabile(OP, S),  
    trasforma(OP, S, NUOVO_S),  
    \+(visitato(NUOVO_S, VISITATI)),  
    solve(NUOVO_S, [S | VISITATI], RESTO).
```

Ricerca depth-first

- **visitato** è un predicato ausiliario che verifica se lo stato raggiunto è già stato visitato (è uguale ad uno appartenente alla lista degli stati visitati):

```
visitato(S, [S1 | _]) :-  
    uguale(S, S1),!.
```

```
visitato(S, [_ | REST]) :-  
    visitato(S, REST).
```

Ricerca breadth-first

- Vantaggi: è completa, è ottimale
- Svantaggi: maggiore complessità (in spazio)

- La strategia di ricerca in ampiezza si ottiene mediante una gestione a coda della lista dei nodi da visitare
- Si determinano tutti gli stati raggiungibili dagli operatori applicabili e li si inserisce in fondo (append) alla lista degli stati da visitare

- To do ... (la soluzione è sul sito del corso)

Ricerca A*

Per realizzare A* si devono mantenere due liste:

- La lista dei nodi da visitare $[S, [PATH, COSTO], STIMA]$ dove:
 - S è un nodo da visitare;
 - $PATH$ è un cammino dallo stato iniziale a S ,
 - $COSTO$ è il costo di tale cammino
 - $STIMA$ è la stima $f^*(S)$ dello stato S su tale cammino;
- che è mantenuta ordinata per valori crescenti della stima dei nodi; ad ogni passo della ricerca si espande quel nodo S per cui la funzione f^* ha valore minimo.
- La lista dei nodi già visitati.

Ricerca A*

- A* non garantisce di trovare la soluzione ottima, dipende dalla funzione euristica:

$$f^*(S) = g^*(S) + h^*(S)$$

- Se la funzione euristica $h^*(n)$ è **ottimistica** ($h^*(n) \leq h(n)$), allora è detta **ammissibile**.
- **Teorema:**
 - Se $h^*(n) \leq h(n)$ per ogni nodo, allora l'algoritmo A* troverà sempre il nodo goal ottimale (in caso di alberi -TREE-SEARCH).
- $f(n)$ non sopravvaluta mai il costo della soluzione che passa per il nodo n

Ricerca A* - euristica

% 4) EVENTUALE EURISTICA

**hstar(S,0):-
 finale(S),!.**

**hstar(S,3-N):-
 my_intersection([on(a,b),on(b,c),
 on_table(c)],S,L),
 length(L,N).**

Ricerca A*

Ad ogni passo si estrae dalla lista dei nodi da visitare lo stato S la cui stima euristica è minima e:

- se lo stato è finale ci si arresta;
- se lo stato non è finale, si determinano tutti i suoi successori. Per ognuno di tali successori NUOVO_S si calcola $f^*(\text{NUOVO_S})$ (sul cammino passante per S) e:
 - se NUOVO_S appartiene alla lista dei nodi da visitare, allora si confronta la nuova stima di NUOVO_S con la vecchia stima e se la nuova stima è minore si sostituisce la vecchia istanza di NUOVO_S dalla lista dei nodi da visitare con la nuova;
 - se NUOVO_S appartiene alla lista dei nodi già visitati, allora si confronta la nuova stima di NUOVO_S con la vecchia stima e se la nuova stima è minore si elimina NUOVO_S dalla lista dei nodi già visitati e lo si riaggiunge alla lista dei nodi da visitare;
 - se NUOVO_S non appartiene a nessuna di tali liste, lo si aggiunge alla lista dei nodi da visitare.

Ricerca A* in Prolog

- **hstar(S,STIMA)**

hstar è la funzione euristica che calcola la stima STIMA per lo stato S; hstar deve essere definita per ogni specifico problema

- **applicabile(OP, PESO, S)**

l'operatore OP con peso PESO è applicabile nello stato S

- **insieme_applicabili_astar(S,OPERATORI)**

OPERATORI è la lista di coppie [OP,PESO] per ogni OP applicabile in S

insieme_applicabili_astar(STATO,OPERATORI) :-

setof([OP,PESO],applicabile(OP,PESO,STATO),OPERATORI),!.

insieme_applicabili_astar(STATO,[]).

- **trasforma(S, OP, NUOVO_S)**

NUOVO_S è lo stato risultante dall'applicazione dell'operatore OP allo stato S

Ricerca A* in Prolog

- **soluzione_astar(SOLUZIONE)**

SOLUZIONE è una sequenza di operatori che costituisce una soluzione del problema definito mediante le relazioni del mondo a blocchi

soluzione_astar(SOLUZIONE):-

iniziale(S),

hstar(S, VALORE),

solve_astar([[S,[],0],VALORE]],[],SOLUZIONE).

Ricerca A* in Prolog

- **solve_astar(DA_VISITARE,VISITATI,SOLUZIONE)**
SOLUZIONE è una soluzione per il problema in cui DA_VISITARE è la lista degli stati da visitare e tenendo conto del fatto che gli stati nella lista VISITATI sono già stati visitati

**solve_astar([[S,[PATH,COSTO],_]|_],_, [PATH,COSTO]):-
finale(S).**

**solve_astar([[S,[PATH,COSTO],STIMA]|REST],VIS,SOL):-
insieme_applicabili_astar(S, OP),
ins_trasf_astar(S,PATH,COSTO,OP,NUOVI),
aggiorna(NUOVI,REST,REST1,VIS,VISITATI1),
solve_astar(REST1,[[S,STIMA]|VISITATI1],SOL).**

Ricerca A* in Prolog

- **ins_trasf_astar(S,PATH,COSTO,OPERATORI,NUOVI)**

"dato lo stato S per cui PATH è un cammino a costo COSTO che collega lo stato iniziale allo stato S, NUOVI è la lista ordinata di triple [NUOVO_S,[[OP|PATH],COSTO1],VAL] per ogni operatore [OP,PESO] in OPERATORI tale per cui NUOVO_S è lo stato che si ottiene applicando l'operatore OP allo stato S e in cui [OP|PATH] è un cammino dallo stato iniziale a NUOVO_S; tale cammino ha costo $COSTO1=COSTO+PESO$; VAL è la stima euristica di NUOVO_S"

Ricerca A* in Prolog

- **aggiorna(ST,DA_VIS,NEW_DA_VIS,VIS,NEW_VIS)**
"NEW_DA_VIS è la nuova lista degli stati da visitare e NEW_VIS è la nuova lista degli stati già visitati ottenute aggiornando DA_VIS e VIS secondo le regole dell'algorithm A* e tenendo conto dei nuovi stati nella lista ST"