Pianificazione in Prolog

Vediamo come risolvere un semplice problema di pianificazione in Prolog utilizzando un pianificatore lineare che effettua una ricerca forward nello spazio degli stati.

Pianificazione nel mondo dei blocchi.

Siano dati:
- un tavolo
- un insieme di blocchi
- un braccio in grado di spostare un blocco alla volta.

Obiettivo:
costruire un piano di operazioni per il braccio che consenta di passare da una data configurazione iniziale dei blocchi sul tavolo ad una configurazione finale.

Dato un dominio con tre blocchi, una possibile configurazione iniziale e finale è riportata nel seguito.

Il problema di pianificazione che si pone come obiettivo lo stato **F** partendo dallo stato **I** si chiama *Anomalia di Sussman*

Per rappresentare lo stato del problema, utilizzeremo un semplice linguaggio (di tipo logico) derivato da quello utilizzato nel sistema STRIPS [*Fikes, Nilsson 71*]

Tale linguaggio è formato dalle seguenti primitive:

- *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).

I due stati di figura sono allora definiti dai due seguenti insiemi (congiunzioni) di asserzioni:

(I) $on\_table(b) \wedge clear(b) \wedge on\_table(a) \wedge on(c,a) \wedge clear(c)$
(F) $on\_table(c) \wedge on(b,c) \wedge on(a,b) \wedge clear(a)$

Azioni possibili per il braccio meccanico:

- *move_to_table(X)*: sposta *X* dalla cima di una pila al tavolo.
- *move_from_table(X,Y)*: sposta *X* dal tavolo alla cima della pila che ha *Y* come cima;
- *move(X,Y):* sposta *X* dalla cima di una pila alla cima di un'altra pila che ha *Y* come cima

Ogni operatore è definito da due parti:

- un insieme di *precondizioni* che indicano in quali stati l'operatore è applicabile.
- un insieme di *azioni* di modifica dello stato.

Descrizione dei tre operatori:

- ***move_to_table(X)***
  precondizioni: *clear(X)*,*on(X,Y)*
  azioni:
  elimina *on(X,Y)*
  aggiungi *on_table(X)*,*clear(Y)*

- ***move_from_table(X,Y)***
  precondizioni: *on_table(X)*,*clear(X)*,*clear(Y)*
  azioni:
  elimina *on_table(X)*,*clear(Y)*
  aggiungi *on(X,Y)*

- ***move(X,Y)***
  precondizioni: *clear(X),on(X,Z),clear(Y)*
  azioni:
  elimina *clear(Y),on(X,Z)*
  aggiungi *on(X,Y),clear(Z)*

La sequenza di operatori che permette di passare dallo stato iniziale (**I**) allo stato finale (**F**) di figura è:

- *move_to_table(c)*,
- *move_from_table(b,c)*,
- *move_from_table(a,b)*

Si parte dal nodo iniziale e il 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 DELLO SPAZIO DEGLI STATI

Nel caso del problema di pianificazione nel mondo dei blocchi può essere adeguata una rappresentazione mediante liste.

Uno stato risulta descritto dalla lista delle asserzioni che lo definiscono.

Tali asserzioni possono a loro volta essere rappresentate come *termini Prolog*.

Ad esempio, gli stati della figura risultano descritti dalle seguenti liste:

(I) [*on_table(b)*, *clear(b)*, *on_table(a)*, *on(c,a)*, *clear(c)*]
(F) [*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"

La relazione *applicabile* permette di definire le precondizioni degli operatori;

Precondizioni: richiedono che certe asserzioni siano vere nello stato cui l'operatore deve essere applicato.

Poiché ogni stato è rappresentato da una lista di asserzioni, verificare se certe asserzioni sono vere corrisponde a verificare se esse appartengono alla lista che definisce lo stato:

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

applicabile(move_from_table(X,Y), S) :-
   member(on_table(X), S),
   member(clear(X), S),
   member(clear(Y), S).

applicabile(move(X,Y), S) :-
   member(clear(X), S),
   member(on(X,Z), S),
   member(clear(Y), S).
```

La relazione *trasforma* permette di definire le azioni.

Poiché ogni trasformazione corrisponde alla eliminazione di un certo insieme di asserzioni ed alla aggiunta di altre asserzioni, si ha la seguente definizione:

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

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

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

E` necessario nel momento in cui si visita uno stato, verificare che tale stato non sia già stato visitato in uno dei passi precedenti del processo di ricerca (grafo).

Per poter effettuare tale verifica è necessario definire una relazione:

uguale(S1, S2)
"S1 e S2 rappresentano lo stesso stato"

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

Si ha allora:

```
uguale([], []).

uguale([X | REST1], S2) :-
   member(X, S2),
   delete1(X, S2, REST2),
   uguale(REST1, REST2).
```

E` necessario introdurre ancora due relazioni per rappresentare lo stato iniziale e lo stato finale.

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"

Nell'esempio precedente si ha:

```
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).
```

Si potrebbe definire come finale un qualunque stato in cui il blocco "a" si trova al di sopra del blocco "b":

```
finale(S) :- member(on(a,b),S).
```

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"

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"

Esercizi proposti:

1. Si riconsideri il problema di pianificazione nel mondo dei blocchi.
   Si supponga che il tavolo abbia dimensioni finite e si ridefiniscano lo stato e gli operatori del problema in tale caso più restrittivo (nella descrizione dello stato sarà importante tener conto delle posizioni dei blocchi su tavolo; nelle precondizioni dell'operatore "move_to_table" sarà necessario verificare che vi sia un posto libero sul tavolo).

2. Si riconsideri il problema di pianificazione nel mondo dei blocchi.
   Si supponga che i blocchi abbiano forma differente (ad esempio che vi siano blocchi di forma cubica ...) e peso differente. Si ridefiniscano lo stato e gli operatori del problema in tale nuovo caso (imponendo condizioni quali ad esempio il fatto che non si può mettere nessun blocco sopra un blocco sferico e non si può mettere un blocco pesante sopra ad uno più leggero).

3. Si descriva il gioco del filetto.

RICERCA IN PROFONDITA`

L'idea dell'algoritmo di ricerca in profondità è la seguente:

- 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.

Si definiscono allora le seguenti relazioni:

soluzione(SOLUZIONE)
"SOLUZIONE è una sequenza di operatori che costituisce una soluzione del problema"

```
soluzione(SOLUZIONE):-
        iniziale(S),
        solve(S, [], SOLUZIONE).
```

solve(S,VISITATI,LISTAdiOP)
"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),
        not visitato(NUOVO_S, VISITATI),
        solve(NUOVO_S, [S | VISITATI], RESTO).
```

visitato(S,VISITATI)
"lo stato S appartiene alla lista degli stati già visitati"

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

RICERCA IN AMPIEZZA

La struttura dell'algoritmo di ricerca in ampiezza è una combinazione delle caratteristiche del programma del mondo a blocchi e dei programmi di ricerca su grafi.

- La lista degli stati da visitare deve essere organizzata come una lista di triple [S,PATH,LISTA_OP] in cui PATH è un cammino dallo stato iniziale allo stato S e LISTA_OP è la lista degli operatori applicati lungo tale cammino; se S è uno stato finale, allora LISTA_OP è un cammino di soluzione.

- Ad ogni passo si seleziona uno stato S nella lista degli stati da visitare e
  - se S è finale ci si arresta;
  - altrimenti si determina l'insieme degli operatori applicabili in S e quindi l'insieme degli stati raggiungibili da S mediante tali operatori.

Tra tali stati, quelli che non appartengono al cammino PATH vengono aggiunti alla lista degli stati da visitare.

```
soluzione_ampiezza(SOLUZIONE):-
   iniziale(S),
   solve_ampiezza([[S, [], []]], SOLUZIONE).

solve_ampiezza([[S,PATH, LISTA_OP] | _ ],LISTA_OP):-
   finale(S).

solve_ampiezza([[S,PATH,LISTA_OP]|REST],LISTA_OP1):-
   finale(S),!,
   solve_ampiezza(REST, LISTA_OP1).

solve_ampiezza([[S,PATH,LISTA_OP]|REST],SOL):-
   ins_applicabili(S, APPLIC),
   ins_trasformati(S,PATH,LISTA_OP,APPLIC,NUOVI),
   append(REST,NUOVI,LISTA),
   solve_ampiezza(LISTA,SOL).
```

La strategia di ricerca in ampiezza è stata ottenuta mediante una gestione a coda della lista dei nodi da visitare.

Occorre definire due nuove relazioni:

ins_applicabili(S, OPERAT)
"OPERAT è l'insieme di operatori applicabili allo stato S"

```
ins_applicabili(S, OPERAT) :-
     setof(OP, applicabile(OP,S), OPERAT), !.

ins_applicabili(S,[]).
```

ins_trasformati(S,PATH,LISTA_OP,APPLIC,NUOVI)
NUOVI=[NUOVO_S,[S|PATH],[OP|LISTA_OP]] è una lista di triple rappresentante gli stati risultanti dall'applicazione degli operatori applicabili APPLIC ad un dato stato S.
In particolare per ogni operatore OP in APPLIC NUOVO_S è lo stato che si ottiene applicando OP a S

```
ins_trasformati(_,_,_,[],[]).

ins_trasformati(S,PATH,LISTA_OP,[OP | REST],ALTRI):-
      trasforma(OP, S, NUOVO_S),
      member(NUOVO_S, PATH),!,
      ins_trasformati(S, PATH, LISTA_OP, REST, ALTRI).

ins_trasformati(S,PATH,LISTA_OP,[OP | REST],
[[NUOVO_S,[S | PATH],[OP | LISTA_OP]]|ALTRI]):-

      trasforma(OP,S,NUOVO_S),
      ins_trasformati(S, PATH, LISTA_OP, REST, ALTRI).
```

Si noti che gli stati che già appartengono al cammino PATH non devono essere presi in considerazione.

La strategia di ricerca in ampiezza è completa (ossia permette di determinare una soluzione, se esiste) mentre la strategia di ricerca in profondità non è completa in quanto può andare in ciclo su un cammino infinito.

Per ovviare a questo problema spesso si associa alla strategia in profondità un limite massimo oltre il quale la ricerca su un cammino non deve essere proseguita.

La strategia in ampiezza permette inoltre di determinare sempre la soluzione di lunghezza minima.

RICERCA EURISTICA

Le strategie di ricerca discusse in precedenza non tengono conto in nessun modo del fatto che vi possono essere cammini più "promettenti" da esplorare.

Utilizzare delle tecniche di ricerca euristica.

Il processo di ricerca euristica può essere schematizzato nel modo seguente:

- Ad ogni passo una parte dello spazio degli stati è stata espansa e vi sono un certo numero di stati (la frontiera della parte espansa del grafo) che devono ancora essere visitati.

- Lo stato da visitare e da espandere viene selezionato considerando lo stato più promettente tra quelli nella frontiera. Se assumiamo che la funzione euristica stimi, per ogni stato S, il costo di una soluzione a partire da S, allora si seleziona quello stato per cui la funzione euristica ha valore minimo.

Il passo più critico nell'effettuare la ricerca euristica è quello della definizione della funzione euristica di stima dei costi di soluzione.

Nel seguito discuteremo un particolare algoritmo di ricerca euristica: **algoritmo A***.

Si supponga che ad ogni operatore sia associato un peso o costo dell'applicazione dell'operatore stesso.

Si introduce quindi una funzione che permette di definire, per ogni stato S, il costo minimo di una soluzione che comprende lo stato S:

f(S) = costo del cammino a costo minimo che collega lo stato iniziale ad uno stato finale passando attraverso lo stato S

In calcolo della funzione **f** richiederebbe di conoscere già la soluzione del problema per cui si cerca di calcolare una sua approssimazione (*funzione di valutazione euristica f**)

f può essere spezzata in due componenti:

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

dove

g(S) = costo del cammino a costo minimo dallo stato iniziale allo stato S

h(S) = costo del cammino a costo minimo da S ad uno stato finale

g(S) può essere approssimata (per eccesso) considerando il cammino a costo minimo che collega lo stato iniziale allo stato S nella parte espansa del grafo;

h(S) deve essere stimata in modo euristico in base alle caratteristiche dello stato S.

Sia *g** la funzione che calcola il cammino a costo minimo sulla parte espansa di grafo dallo stato iniziale ad uno stato S (per cui $g^*(S)>g(S)$) e sia *h** una funzione di stima della funzione *h*.

La *funzione di valutazione euristica f** per la ricerca utilizzata dall'algoritmo A* è:

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

Ad ogni passo della ricerca si espande quel nodo S per cui la funzione *f** ha valore minimo.

Si dice che una strategia di ricerca è *ammissibile* se essa determina sempre la soluzione ottimale (quando esiste una soluzione).

Si può dimostrare che se *h** è una stima per difetto di *h* (ossia se, per ogni S, $h^*(S) \leq h(S)$) allora l'algoritmo A* è ammissibile.

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;

  La lista dei nodi da visitare viene mantenuta ordinata per valori crescenti della stima dei nodi.

- La lista dei nodi già visitati.

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*(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.

Realizzazione in Prolog:

Supporremo di avere a disposizione le seguenti relazioni:

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 operatore 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"

Si ha allora il seguente programma:

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).
```

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).
```

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"

```
ins_trasf_astar(_, _, _, [], []).

ins_trasf_astar(S,PATH,COSTO,[[OP,PESO]|REST_OP],
[[NUOVO_S,[[OP | PATH],COSTO1],VAL]| REST_ST]) :-
        trasforma(OP, S, NUOVO_S),
        COSTO1 is COSTO+PESO,
        hstar(NUOVO_S,STIMA),
        VAL is COSTO1 + STIMA,
ins_trasf_astar(S,PATH,COSTO,REST_OP,REST_ST).
```

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'algoritmo A* e tenendo conto dei nuovi stati nella lista ST"

```
aggiorna([], DA_VIS, DA_VIS, VIS, VIS).

aggiorna([[S,[PATH,COSTO],STIMA]|REST],DA_VIS,
NEW_DA_VIS, VIS, NEW_VIS):-

    member([S, STIMA1], VIS),
    STIMA1=<STIMA, !,
    aggiorna(REST,DA_VIS,NEW_DA_VIS,VIS,NEW_VIS).

aggiorna([[S,[PATH,COSTO],STIMA]|REST],DA_VIS,
NEW_DA_VIS, VIS, NEW_VIS):-

   member([S, STIMA1], VIS),
   STIMA1 > STIMA, !,
   delete1([S, STIMA1], VIS, VIS1),
   insert([S, [PATH,COSTO], STIMA],DA_VIS,DA_VIS1),
   aggiorna(REST,DA_VIS1,NEW_DA_VIS,VIS1,NEW_VIS).

aggiorna([[S,[PATH,COSTO],STIMA]|REST],DA_VIS,
NEW_DA_VIS, VIS, NEW_VIS):-

     member([S, _, STIMA1], DA_VIS),
     STIMA1 =< STIMA, !,
     aggiorna(REST,DA_VIS,NEW_DA_VIS, VIS, NEW_VIS).

aggiorna([[S,[PATH,COSTO],STIMA]|REST],DA_VIS,
NEW_DA_VIS, VIS, NEW_VIS):-

    member([S, _, STIMA1], DA_VIS),
    STIMA1 > STIMA, !,
    delete1([S, _, STIMA1], DA_VIS, DA_VIS1),
    insert([S,[PATH,COSTO],STIMA],DA_VIS1, DA_VIS2),
    aggiorna(REST,DA_VIS2,NEW_DA_VIS,VIS,NEW_VIS).

aggiorna([[S,[PATH,COSTO],STIMA]|REST],DA_VIS,
NEW_DA_VIS, VIS, NEW_VIS):-

    insert([S, [PATH,COSTO], STIMA], DA_VIS, DA_VIS1),
    aggiorna(REST,DA_VIS1,NEW_DA_VIS,VIS,NEW_VIS).
```

insert([S, X, STIMA], [], [[S, X, STIMA]]).

```
insert([S,X,STIMA], [[S1,X1,STIMA1]|REST], [[S,X,STIMA],
[S1,X1,STIMA1]|REST]):-

    STIMA =< STIMA1, !.

insert([S,X,STIMA],[[S1,X1,STIMA1]|REST],
[[S1,X1,STIMA1]|NEW_REST]):-

     insert([S,X,STIMA], REST, NEW_REST).
```

Esercizi proposti

I. Valutare in base a diversi valori di f* il comportamento dell'algoritmo di ricerca euristica. Studiare il comportamento anche in relazione all'Anomalia di Sussmann.

II. Realizzare in Prolog l'algoritmo backward con regression completo (che prova tutti gli ordinamenti possibili dei goal e dei loro sottogoal) .

III. Realizzare in Prolog l'algoritmo (Strips-like) che considera i goal separatamente.

Soluzione Esercizio I

Per semplicità a tutti gli operatori è assegnato lo stesso peso (= 1).

```
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),

applicabile(move(X,Y), 1, S) :-
        member(clear(X), S),
        member(on(X,Z), S),
        member(clear(Y), S),
```

La relazione *hstar(S,STIMA)* che calcola la stima *STIMA* per lo stato *S* è stata definita assegnando ad ogni stato *S* un costo pari al numero di blocchi che in *S* si trovano in una posizione diversa da quella desiderata. Un blocco si trova nella posizione desiderata quando l'oggetto sopra il quale si trova coincide con quello indicato dal goal.

Tenendo conto che il problema da risolvere è l'Anomalia di Sussmann con:

- numero di blocchi totali: 3
- disposizione finale: a su b, b su c, c sul tavolo

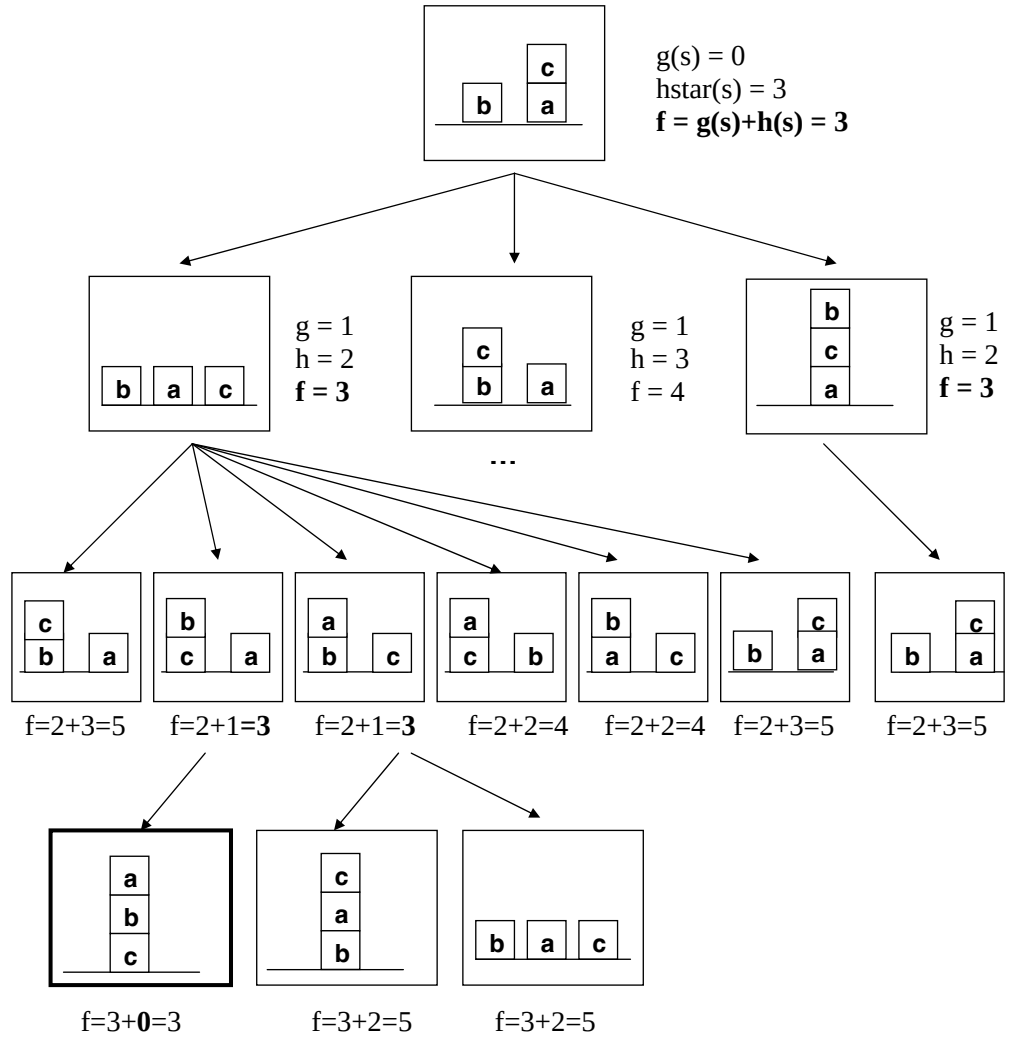
si ha:

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

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

Si noti che l'euristica così definita è un'euristica ammissibile visto che con una mossa si può al più mettere un blocco nella posizione desiderata.

ALBERO DI RICERCA



Vediamo alcuni passi di trace prodotti dall'algoritmo di ricerca euristica (con relazione *hstar* sopra definita) applicato al problema dell'Anomalia di Sussman

(1) 0 CALL *soluzione_astar(Sol)* (dbg)?- creep

(3) 1 CALL *hstar*([on_table(a), on_table(b), clear(b), on(c, a), clear(c)],VALORE) (dbg)?- skip
(3) 1 EXIT *hstar*([on_table(a), on_table(b), clear(b), on(c, a), clear(c)],3 - 0) (dbg)?- creep

(8) 1 CALL *solve_astar*([[[on_table(a), on_table(b), clear(b), on(c, a), clear(c)], [[], 0], 3 - 0]], [], S) (dbg)?-

(9) 2 CALL *finale*([on_table(a), on_table(b), clear(b), on(c, a), clear(c)]) (dbg)?- creep
(9) 2 FAIL *finale*([on_table(a), on_table(b), clear(b), on(c, a), clear(c)]) (dbg)?- creep

(8) 1 NEXT *solve_astar*([[[on_table(a), on_table(b), clear(b), on(c, a), clear(c)], [[], 0], 3 - 0]], [], S) (dbg)?-

(88) 2 CALL *solve_astar*([[[on(b, c), on_table(a), clear(b), on(c, a)], [[move_from_table(b, c)], 1], 3], [[on_table(c), clear(a), on_table(a),on_table(b),clear(b), clear(c)], [[move_to_table(c)], 1], 3], [[on(c, b), clear(a), on_table(a), on_table(b), clear(c)], [[move(c, b)], 1], 4]], [[[on_table(a), on_table(b), clear(b), on(c, a), clear(c)], 3 - 0]], S) (dbg)?- creep

(89) 3 CALL *finale*([on(b, c), on_table(a), clear(b), on(c, a)]) (dbg)?- skip

(89) 3 FAIL *finale*([on(b, c), on_table(a), clear(b), on(c, a)]) (dbg)?- creep
(88) 2 NEXT *solve_astar*([[[on(b, c), on_table(a), clear(b), on(c, a)], [[move_from_table(b, c)], 1], 3], [[on_table(c), clear(a), on_table(a), on_table(b), clear(b), clear(c)], **[[move_to_table(c)], 1], 3]**, [[on(c, b), clear(a), on_table(a), on_table(b), clear(c)], [[move(c, b)], 1], 4]], [[[on_table(a), on_table(b), clear(b), on(c, a), clear(c)], 3 - 0]], S) (dbg)?-

(126) 3 CALL *solve_astar*([[[on_table(c), clear(a), on_table(a), on_table(b), clear(b), clear(c)], [[move_to_table(c)], 1], 3], [[on(c,b),clear(a),on_table(a),on_table(b),clear(c)], [[move(c, b)], 1], 4], [[on_table(b),clear(c),on_table(a),clear(b),on(c, a)], [[move_to_table(b), move_from_table(b, c)], 2], 5]], [[[on(b, c), on_table(a), clear(b), on(c, a)], 3], [[on_table(a), on_table(b), clear(b), on(c, a), clear(c)], 3 - 0]], S) (dbg)?-

(127) 4 CALL *finale*(…)
(127) 4 FAIL *finale*(…)

(126) 3 NEXT *solve_astar*(…)

(292) 4 CALL *solve_astar*
([[[on(b,c),on_table(c),clear(a),on_table(a),clear(b)],
[[move_from_table(b,c), move_to_table(c)],2],3],
[[on(a,b),on_table(c),clear(a),on_table(b),clear(c)],
[[move_from_table(a, b), move_to_table(c)], 2], 3], …)

(293) 5 CALL *finale*([on(b, c), on_table(c), clear(a), on_table(a), clear(b)]) (dbg)?- skip

(293) 5 FAIL *finale*([on(b, c), on_table(c), clear(a), on_table(a), clear(b)]) (dbg)?-

(292) 4 NEXT solve_astar([[[on(b, c), on_table(c), clear(a), on_table(a), clear(b)], [[move_from_table(b,c), move_to_table(c)], 2], 3], [[on(a, b), on_table(c), clear(a), on_table(b), clear(c)],…) (dbg)?-

(395) 5 CALL solve_astar([[[on(a, b), on(b, c), on_table(c), clear(a)], **[[move_from_table(a, b), move_from_table(b, c), move_to_table(c)], 3], 3]**, [[on(a,b), on_table(c), clear(a), on_table(b), clear(c)], …) (dbg)?-

(396) 6 CALL *finale*([on(a, b), on(b, c), on_table(c), clear(a)]) (dbg)?- skip
(396) 6 *EXIT *finale*([on(a, b), on(b, c), on_table(c), clear(a)]) (dbg)?- creep

(395) 5 *EXIT *solve_astar*(…)
(292) 5 *EXIT *solve_astar*(…)
(126) 5 *EXIT *solve_astar*(…)
(88) 2 * EXIT *solve_astar*(…)
(8) 2 * EXIT *solve_astar*(…)

(1) 0 *EXIT *soluzione_astar*([[move_from_table(a, b), move_from_table(b, c),move_to_table(c)], 3]) (dbg)?- creep

Sol = [[move_from_table(a, b), move_from_table(b, c), move_to_table(c)], 3]
More? (;)
Nota: l'Anomalia di Sussman non si è verificata!!!