Relazione per gli esercizi PROLOG e CLINGO

per il corso di Intelligenza Artificiale e Laboratorio

Andrea Latella, Roberto Pesando, Davide Furno

16 aprile 2015

Indice

Parte I

Prolog

Capitolo 1

Strategie non informate

In questo capitolo verrà trattata l' implementazione delle due strategie non informate viste a lezione ovvero la ricerca in ampiezza e la ricerca in profondità; per ognuna di queste strategie verrà proposta l'implementazione generale e l'applicazione specifica per due domini che sono il dominio dei cammini e il dominio del mondo dei blocchi.

1.1 Ricerca in ampiezza

1.1.1 Il codice sviluppato

```
1  ric_amp([nodo(S,LISTA_AZ)|_],LISTA_AZ):- finale(S).
2  ric_amp([nodo(S,LISTA_AZ)|RESTO],SOL):-
3          num_nodi_open,
4          espandi(nodo(S,LISTA_AZ),LISTA_SUCC),
5          append(RESTO,LISTA_SUCC,CODA),
6          ric_amp(CODA,SOL).
7
8
9  espandi(nodo(S,LISTA_AZ),LISTA_SUCC):-
10         findall(AZ,applicabile(AZ,S),AZIONI),
11
12         successori(nodo(S,LISTA_AZ),AZIONI,LISTA_SUCC).
13
14 successori(_,[],[]).
15 successori(nodo(S,LISTA_AZ),[AZ|RESTO],[nodo(NUOVO_S,NUOVA_LISTA_AZ)|ALTRI]):-
16         trasforma(AZ,S,NUOVO_S),
17         append(LISTA_AZ,[AZ],NUOVA_LISTA_AZ),
18         successori(nodo(S,LISTA_AZ),RESTO,ALTRI).
19
20 num_nodi_open:-
21         nb_getval(counter, N1),
22         New1 is N1 + 1,
23         nb_setval(counter, New1).
24
25
26 ampiezza:- iniziale(S),
27         nb_setval(counter , 0),
28         ric_amp([nodo(S,[])],SOL),
29         writeln(SOL),
```

```
30          write(N_res).
```

1.1.2 Analisi dettagliata della strategia

La ricerca in ampiezza è stata implementata in modo classico, fornendo una regola ric_amp([nodo (S,LISTA_AZ)|_],LISTA_AZ) per il caso base e una ric_amp ([nodo (S,LISTA_AZ)|RESTO],SOL) per il caso generico. La prima non fa altro che controllare che lo stato S sia lo stato finale, mentre la seconda invoca le regole espandi(nodo(S,LISTA_AZ),LISTA_SUCC) e append(RESTO, LISTA_SUCC,CODA), per poi richiamare ricorsivamente ric_amp. La regola espandi non fa altro che prendere lo stato s, espanderlo e cercare tutte le azioni applicabili(con **findall** da quello stato; tramite la regola successori() costruisce la lista dei nuovi stati disponibili a partire dallo stato s, che saranno poi aggiunti agli stati ancora da espandere tramite la regola append(). Infine richiama ricorsivamente la regola per controllare di aver raggiunto lo stato finale e in caso contrario proseguire con l'espansione dei nodi. Se la soluzione si trova a una profondità d e il branching factor è b verranno espansi $b^{d+1} - b$. Tutti questi nodi dovranno rimanere in memoria. Dato il numero elevato di nodi espansi e che un nodo può essere espanso più volte questa strategia può non terminare, perchè fa raggiungere il limite di memoria disponibile.

1.1.3 Dominio dei Cammini

Per quanto riguarda il dominio dei cammini, la ricerca in ampiezza risulta sensata solo se la soluzione si trova ad un livello di profondità basso, altrimenti si incorre nell'esaurimento di memoria. Per quanto riguarda l'esempio specifico fornito dal Prof. Martelli e quello del professor Torasso la ricerca della soluzione ha esito negativo perchè la soluzione si trova ad un livello di profondità che la ricerca in ampiezza non è in grado di raggiungere causando un esaurimento della memoria.

1.1.4 Dominio del Mondo Dei Blocchi

Anche per quanto riguarda il dominio del mondo dei blocchi il risultato non cambia. Infatti anche in questo caso l'esempio fornito dal professore ha esito negativo perchè nuovamente la soluzione risulta troppo in profondità per essere raggiunta.

1.2 Ricerca in ampiezza su grafi

1.2.1 Il codice sviluppato

```
1  ampiezza :-
2      iniziale(S),
3      finale(Goal),
4      nb_setval(counter , 0),
5      time(ric_ampiezza([nodo(S, [])], [], Ris)),
6      writeln(Ris),
7      nb_getval(counter, N_res),
8      write(N_res).
9
10
11 ric_ampiezza([nodo(S, Lista_Az)|_], _, Lista_Az) :- finale(S), !.
12 ric_ampiezza([nodo(S, Lista_Az) | R_lista_open], Closed, Lista_Ris) :-
13         member(S, Closed) ->
14                 ric_ampiezza(R_lista_open, Closed, Lista_Ris);
15         num_nodi_open,
16         open_node(nodo(S, Lista_Az), Lista_children),
17         ord_union(Lista_children, R_lista_open, Nuova_open),
18         ric_ampiezza(Nuova_open,[S|Closed], Lista_Ris).
19
20 open_node(nodo(S, Lista_Az), Lista_childern) :-
21         findall(Az, applicabile(Az,S), Az_applicabili),
22         node(nodo(S, Lista_Az), Az_applicabili, Lista_childern).
23
24 num_nodi_open:-
25         nb_getval(counter, N1),
26         New1 is N1 + 1,
27         nb_setval(counter, New1).
28
29 node(_,[],[]).
30 node(nodo(S, Lista_Az), [Az|R_az], Lista_children) :-
31         finale(Goal),
32         trasforma(Az, S, Nuovo_S),
33         append(Lista_Az, [Az], Nuova_lista_az),
34         node(nodo(S, Lista_Az), R_az, Old_children),
35         ord_add_element(Old_children, nodo(Nuovo_S, Nuova_lista_az), Lista_children).
```

1.2.2 Analisi dettagliata della strategia

La ricerca in ampiezza su grafi può essere considerata come un ottimizzazione della ricerca in ampiezza classica. Introduce il concetto di lista chiusa per migliorare la ricerca della soluzione. Una lista chiusa non è altro che una lista composta da tutti i nodi visitati. Quando dobbiamo espandere un nuovo nodo andiamo prima a verificare che questo nodo non sia nella lista dei nodi chiusi (nel codice con `member(S, Closed)-> ric_ampiezza(R_lista_open , Closed, Lista_Ris);`); se appartiene a quella lista lo scartiamo, se invece non appartiene a quella lista allora possiamo procedere con l'espansione con la regola `open_node(nodo(S, Lista_Az), Lista_children),`. Evitando di ri-espandere nodi già completamente visitati migliora di molto la versione base dell'algoritmo della visita in ampiezza, riducendo di molto il numero di nodi che vengono effettivamente visitati, e di conseguenza lo spazio di memoria necessario per

contenerli. Una volta aperto il nodo non si fa altro che verificare tutte le azioni applicabili da quel nodo **findall**(Az, applicabile(Az,S), Az_applicabili),, dopodiche si invoca la regola node(nodo(S, Lista_Az), Az_applicabili, Lista_childern) che non fa altro che applicare le azioni prese dalla lista delle azioni possibile ed inserire i nodi risultanti nei nodi da visitare nei passi successivi.

1.2.3 Dominio dei Cammini

Vista l'ottimizzazione apportata all'algoritmo di base per la ricerca in ampiezza, il nuovo algoritmo è in grado di trovare una soluzione agli esempi presi in esame precedentemente con la sola ricerca in ampiezza. Nello specifico sull'esempio del professor Martelli (cammini 10x10) l'algoritmo trova una soluzione in 0.017 secondi, visitando appena 66 nodi e facendo 19508 inferenze. Si è riusciti a passare dalla non terminazione ad avere una soluzione con appena 66 nodi visitati. Anche per quanto riguarda l'esempio del professor Torasso (cammini 20x20) l'algoritmo è in grado di trovare una soluzione in 0.031 secondi, visitando 374 nodi e facendo 178644 inferenze. Anche in questo caso abbiamo un aumento impressionante delle prestazioni, dato che ovviamente nella versione di base non era in grado di generare una soluzione. Qui di seguito abbiamo i grafici riassuntivi dei risultati ottenuti.

Figura 1.1: Grafici per il dominio di cammini.

1.2.4 Dominio del Mondo Dei Blocchi

Anche in questo caso la strategia è in grado di fornirci una soluzione a differenza della controparte di base. Per quanto riguarda l'esempio fornito dal professor Martelli l'algoritmo è in grado di generare una soluzione in 0.085 secondi, visitando 487 nodi e facendo 528177 inferenze. Per quanto riguarda l'esempio del professor Torasso la situazione non cambia dalla versione base; l'algoritmo non è in grado di trovare la soluzione in quanto lo spazio di ricerca rimane troppo vasto per esser esplorato tutto. Qui di seguito abbiamo i grafici riassuntivi dei risultati ottenuti.

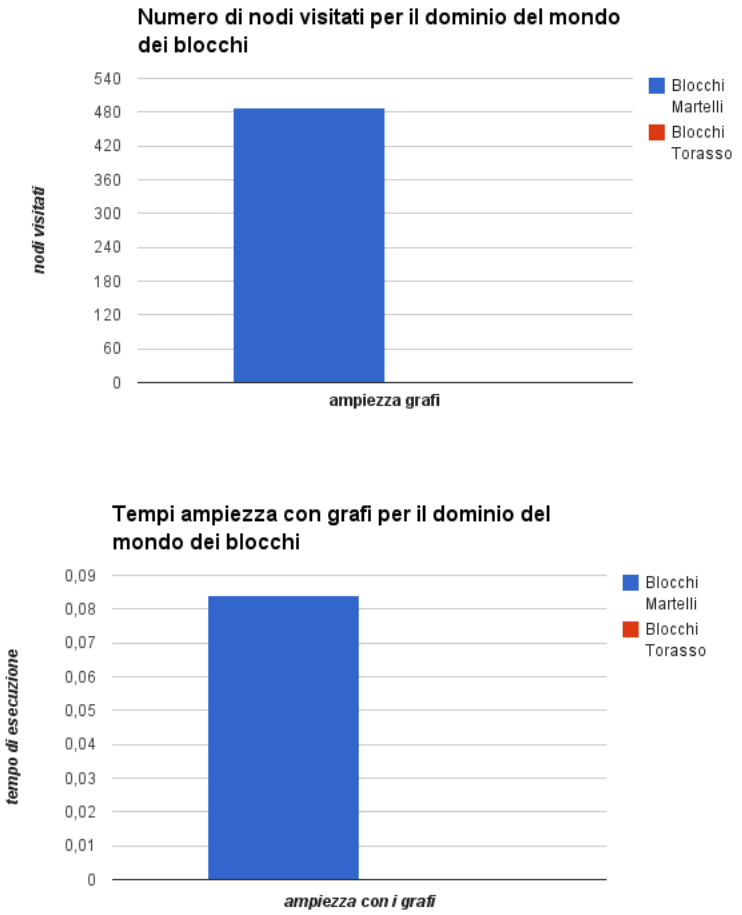

Figura 1.2: Grafici per il dominio dei blocchi.

1.3 Ricerca in profondità

1.3.1 Il codice sviluppato

```
1  ric_prof_cc_lim(S,_,_,[]) :- finale(S),!.
2  ric_prof_cc_lim(S,D,Visitati,[Az|Resto]) :-
3      D>0,
4      applicabile(Az,S),
5      trasforma(Az,S,Nuovo_S),
6      \+ member(Nuovo_S,Visitati),
7      num_nodi_open,
8      D1 is D-1,
9      ric_prof_cc_lim(Nuovo_S,D1,[S|Visitati],Resto).
10
11 ric_prof_cc_id(I,D,Ris) :- ric_prof_cc_lim(I,D,[],Ris).
12 ric_prof_cc_id(I,D,Ris) :-
13     D1 is D+1,
14     ric_prof_cc_id(I,D1,Ris).
15
16 num_nodi_open:-
17         nb_getval(counter, N1),
18         New1 is N1 + 1,
19         nb_setval(counter, New1).
20
21 prof_lim(D) :- iniziale(I),ric_prof_cc_lim(I,D,[],Ris),write(Ris).
22 prof_id :-
23         iniziale(I),
24         nb_setval(counter , 0),
25         time(ric_prof_cc_id(I,1,Ris)),
26         nb_getval(counter, N_res),
27         writeln(Ris),
28         write(N_res),
29         write('\n').
```

1.3.2 Analisi dettagliata della strategia

La ricerca in profondità è stato sviluppata in 2 versione; la prima riceve in input una profondità D oltre la quale la ricerca non può andare; la seconda invece parte con una profondità di 1 e man mano che vengono visitati tutti i nodi entro quella profondità senza trovare soluzione, la profondità viene aumentata di un livello. La seconda procedura è denominata Iterative Deepening. L'implementazione della prima versione è molto semplice. Come per la ricerca in ampiezza abbiamo 2 regole, una per il caso base e una per il caso generico. Nuovamente la regola per il caso base non fa altro che verificare che lo stato corrente `S` sia lo stato finale. La regola del passo generico è stata sviluppata nel seguente modo: se la profondità non è minore di zero applico allo stato corrente S la prima delle azioni applicabili disponibili, dopodiche controllo che il nuovo stato non sia stato già visitato, decremento la profondità di 1 ed infine richiamo la regola ricorsivamente sul nuovo stato. Se la profondità risulta minore di zero, significa che ho superato il limite imposto e di conseguenza non posso scendere oltre nell'espansione. Prolog farà backtracking e procederà con l'applicazione di un'altra delle azioni ap-

plicabili per quel nodo e si procede nuovamente come descritto sopra. Se la soluzione è oltre il limite imposto a priori Prolog resituirà false, altrimenti viene restituita la lista delle azioni per raggiungerla.

L'implementazione della seconda versione sfrutta la prima. Utilizza le due regole per scendere in profondità nei vari stati, ai quali viene aggiunta una regola per aumentare il limite di un livello, quando la ricerca non trova soluzione all'interno del limite corrente. Il limite che viene fornito a priori è di 1 e ci si ferma solo quando si trova la soluzione. La ricerca in profondità ha requisiti di memoria molto bassi, in quanto deve ricordare tutti i nodi di un solo cammino insieme a tutti i fratelli non espansi di ogni nodo del cammino. Una volta espanso e che tutti i suoi discendenti sono stati trovati il nodo può esser rimosso dalla memoria. A differenza della ricerca in ampiezza il numero massimo di nodi da mantenere in memoria con un fattore di ramificazione b e profondità massima m è $bm + 1$.

1.3.3 Dominio dei Cammini

A differenza della ricerca in ampiezza, la ricerca a profondità limitata con approfondimento iterativo (iterative deepening) è in grado di fornire una soluzione ottima e di lunghezza minima per i problemi relativi al dominio dei cammini, dato che tutte le azioni relative a questo dominio hanno costo unitario. Per quanto riguarda l'esempio specifico fornito dal Prof. Martelli (cammino 10x10) la ricerca in profondità limitata con approfondimento iterativo è in grado di trovare una delle soluzioni di lunghezza minima (se si hanno più soluzioni con la stessa lunghezza possono essere visualizzate tramite il comando ; dopo la visualizzazione della prima soluzione). La soluzione viene trovata in tempo ragionevole, circa 3.203 secondi con un numero di nodi visitati pari a 548376 e un numero di inferenze pari a 21265407. Per quanto riguarda l'esempio fornito dal professor Torasso (cammino 20x20), la ricerca in profondità impiega un tempo taltmente elevato che ci è stato impossibile verificare la sua terminazione. Le nostre prove sul dominio sono durate 30 e 50 minuti senza riuscire ad ottenere una soluzione, di conseguenza possiamo affermare che non è una strategia ragionevole per risolvere quel determinato esempio. Qui di seguito abbiamo i grafici riassuntivi dei risultati ottenuti.

Figura 1.3: Grafici per il dominio dei cammini.

1.3.4 Dominio del Mondo dei Blocchi

Anche in questo caso la strategia implementata è in grado di fornire la soluzione ottima a lunghezza minima. Per quanto riguarda l'esempio del professor Martelli l'algoritmo trova soluzione in 0.459 secondi circa, visitando 13222 nodi e facendo 3364820 inferenze. Invece sull'esempio fornito dal professor Torasso l'algoritmo si comporta decisamente peggio ma riesce comunque a trovare la soluzione in un tempo ragionevole che è di 232.268 secondi. Ovviamente visto l'elevato tempo richiesto per trovare la soluzione l'algoritmo ha visitato 4180656 nodi e ha compiuto 1548207210 inferenza. Qui di seguito abbiamo i grafici riassuntivi dei risultati ottenuti.

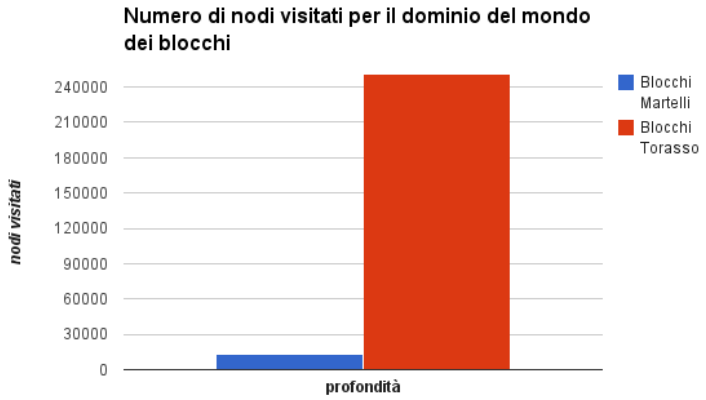

Figura 1.4: Nodi visitati per i domini del mondo dei blocchi

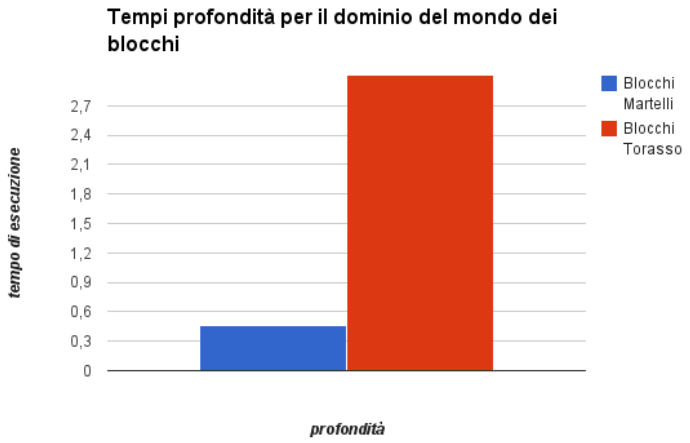

Figura 1.5: Tempi di calcolo per i domini del mondo dei blocchi

Capitolo 2

Strategie informate

In questo capitolo verrà trattata l'implementazione delle strategie informate viste: la ricerca in profondità ad approfondimento iterativo con stima (IDA*) e la ricerca in ampiezza sui grafi con stima (A*). Per ognuna di queste strategie verrà proposta l'implementazione generale e l'euristica specifica utilizzata per il dominio dei cammini, il dominio del mondo dei blocchi,il dominio della metropolitana.

2.1 Euristiche Utilizzate

2.1.1 Euristica dei cammini

```
1  calcolo_euristica(pos(R,C),pos(R1,C1), G) :-
2          X is R - R1,
3          Y is C - C1,
4          abs(X, Xabs),
5          abs(Y, Yabs),
6          H is Xabs + Yabs,
7          F is G + H,
8          retract(f_val(_)),
9          assert(f_val(F)),
10         retract(h_val(_)),
11         assert(h_val(H)).
```

Per quanto riguarda il dominio dei cammini l'euristica utilizzata per generare i valori di `f_val` è stata la classica distanza di Manhattan. Vengono presi in input le coordinate `x` e `y` del nodo corrente e del nodo goal e si esegue questo semplice calcolo: $L1(p1, p2) = |x1 - x2| + |y1 - y2| F = G + L1$

`G` viene passata come parametro nella regola, mentre il valore `F` che viene calcolato viene asserito come `f_val`.

2.1.2 Euristica dei blocchi locale

```
1  calcolo_euristica(S, G) :-
2          goal(Res),
```

```
3           ord_subtract(S, Res, Diff),
4           calcola_h(Diff, Ndiff),
5           h_val(Ndiff),
6           F is G + Ndiff,
7           retract(f_val(_)),
8           assert(f_val(F)).
9
10  calcola_h([],0).
11  calcola_h([_|Resto_Ok],H) :-
12          calcola_h(Resto_Ok, H1),
13          H is H1 + 1,
14          retract(h_val(_)),
15          assert(h_val(H)).
```

Le euristiche utilizzate per il mondo dei blocchi hanno richiesto delle considerazioni più complesse rispetto all'euristica del mondo dei cammini. L'idea che ha guidato l'euristica locale è stata quella di stimare il costo dallo stato attuale allo stato finale come la differenza di fatti tra questi due stati. L'avvicinarsi dello stato attuale a quello finale è stato, quindi, rendere l'insieme dei fatti dello stato attuale sempre più simile a quello dello stato finale, finendo poi per farli coincidere trovata una soluzione. L'euristica utilizzata fornisce il nostro valore di f_val sommando il costo del cammino trovato dal nodo iniziale al nodo corrente, con la differenza insiemistica tra l'insieme degli elementi dello stato attuale e quello dello stato finale. Questo viene fatto tramite ord_subtract(List1,List2,Res). Ord_subtract è una regola nativa di Prolog e non fa altro che sottrarre la List1 dalla List2 due e mettere la lista risultante in Res. Infine tramite la regola calcola_h contiamo gli elementi di Res. Si cerca di trovare la mossa che più assottilia la differenza tra i due insiemi, fino al raggiungimento della parificazione degli stessi.

2.1.3 Euristica dei blocchi globale

Nella progettazione della prima euristica ci siamo accorti che l'euristica creata potrebbe non esser una buona euristica. Spieghiamo il perchè con il seguente esempio:

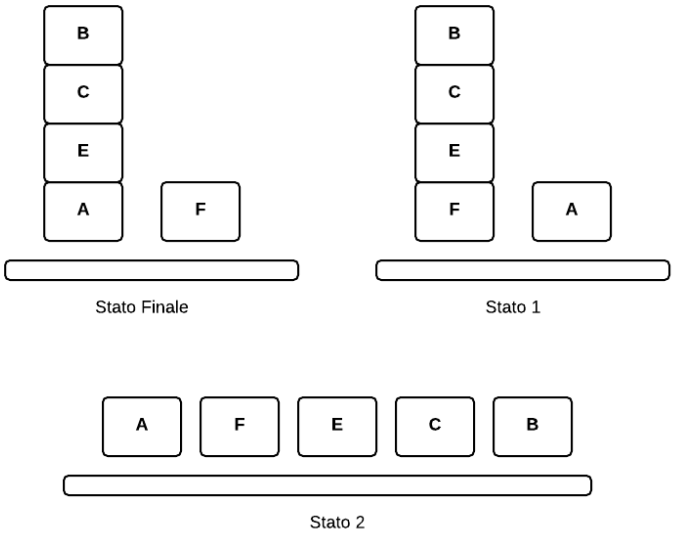

Figura 2.1: Esempio Mondo dei Blocchi

Se dovessimo valutare se è preferibile espandere lo stato 1 o lo stato 2 per arrivare allo stato finale, la risposta, anche in maniera molto intuibile, sarebbe dallo stato 2. Questo perchè pur essendoci dei blocchi nello stato 1 che in una visione locale sono posizionati in maniera corretta rispetto allo stato finale (ad. esempio il blocco f è ontable, il blocco c è sopra il blocco e, ecc...) in una visione globale dello stato questo non è confermato. Valutando la pila b-c-e-f pur avendo 3/4 blocchi in posizione corretta, per arrivare a una soluzione corretta bisogna smontare l'intera pila e ricostruirla. L'euristica globale per ogni blocco non ontable controlla che l'intero supporto del blocco sia corretto rispetto al supporto dello stato finale.In questo caso si ha un punteggio di 1 x Nr blocchi del supporto, altrimenti -1 x Nr blocchi del supporto.

Per l'esempio precedente riportiamo i punteggi per ogni blocco.

Stato 2:

```
1 blocco A = B = C = E = F =0
2
3 Totale = 0
```

Tutti i blocchi hanno punteggio 0 perchè sono ontable. L'idea che vi è dietro è che anche se un blocco si trova in una posizione sbagliata, ma è in una

situazione in cui è immediatamente spostabile allora non si va ad assegnargli un punteggio negativo, perchè siamo in una condizione che ci permette di passare a costruire un pezzo della soluzione. Mentre se si deve distruggere qualcosa di precedentemente creato allora si ha una penalità.

Stato 1:

```
blocco A = 0
blocco B = −3
blocco C = −2
blocco E = −1
blocco F = 0

Totale = −6
```

Di seguito riportiamo il codice per tale euristica.

```
calcolo_euristica(S, G) :−
        goal(Goal),
        list_block(B),
        retract(cost(_)),
        assert(cost(0)),
        cost_of_state(S,Goal,B),
        cost(Ris),
        F is G − Ris,
        retract(f_val(_)),
        assert(f_val(F)).

cost_of_state(S,Goal,[]).

cost_of_state(S,Goal,[H|T]) :− block(H),
                                \+ ord_memberchk(ontable(H),S),
                                check_pila(S,Goal,H,0,0),
                                cost_of_state(S,Goal,T).

cost_of_state(S,Goal,[H|T]) :− block(H),
                                ord_memberchk(ontable(H),S),
                                cost_of_state(S,Goal,T).

cost_of_state(S,Goal,[H|T]) :− block(H),
                                ord_memberchk(holding(H),S),
                                cost_of_state(S,Goal,T).

check_pila(S,Goal,Blocco,N_ok,N_nok) :−
    ord_memberchk(ontable(Blocco),S),
    ord_memberchk(ontable(Blocco),Goal),
    ord_memberchk(clear(Blocco),S),
    ord_memberchk(clear(Blocco),Goal).

check_pila(S,Goal,Blocco,N_ok,N_nok) :−
    ord_memberchk(ontable(Blocco),S),
    \+ ord_memberchk(ontable(Blocco),Goal),
    cost(R),
    retract(cost(_)),
    R1 is R − (N_ok + N_nok),
    assert(cost(R1)).

check_pila(S,Goal,Blocco,N_ok,N_nok) :−
    ord_memberchk(ontable(Blocco),S),
    ord_memberchk(ontable(Blocco),Goal),
    block(X),
    ord_memberchk(on(X,Blocco),S),
```

```
46      ord_memberchk(on(X,Blocco),Goal),
47      N_nok > 0,
48      cost(R),
49      retract(cost(_)),
50      R1 is R − (N_ok + N_nok),
51      assert(cost(R1)).
52
53  check_pila(S,Goal,Blocco,N_ok,N_nok) :−
54      ord_memberchk(ontable(Blocco),S),
55      ord_memberchk(ontable(Blocco),Goal),
56      block(X),
57      ord_memberchk(on(X,Blocco),S),
58      \+ ord_memberchk(on(X,Blocco),Goal),
59      cost(R),
60      retract(cost(_)),
61      R1 is R − (N_ok + N_nok),
62      assert(cost(R1)).
63
64  check_pila(S,Goal,Blocco,N_ok,N_nok) :−
65      ord_memberchk(ontable(Blocco),S),
66      ord_memberchk(ontable(Blocco),Goal),
67      block(X),
68      ord_memberchk(on(X,Blocco),S),
69      ord_memberchk(on(X,Blocco),Goal),
70      N_nok =:= 0,
71      cost(R),
72      retract(cost(_)),
73      R1 is R + (N_ok + N_nok),
74      assert(cost(R1)).
75
76  check_pila(S,Goal,Blocco,N_ok,N_nok) :−
77      block(X),
78      ord_memberchk(on(Blocco,X),S),
79      \+ ord_memberchk(on(Blocco,X),Goal),
80      N is N_nok + 1,
81      check_pila(S,Goal,X,N_ok,N).
82
83  check_pila(S,Goal,Blocco,N_ok,N_nok) :−
84      block(X),
85      ord_memberchk(on(Blocco,X),S),
86      ord_memberchk(on(Blocco,X),Goal),
87      N is N_ok + 1,
88      check_pila(S,Goal,X,N,N_nok).
```

2.1.4 Euristica della metropolitana di Londra

Il dominio della metropolitana di Londra risulta leggermente differente dai due domini presi in esame precedentemente, perchè le azioni non hanno un costo unitario. Oltre a dover sviluppare un euristica adatta a calcolare un f_val() per poter eseguire la ricerca, bisogna anche far fronte che non basta più un semplice incremento di uno del nostro g_val dopo ogni azione, ma serve un incremento diverso in base all'azione che stiamo per intraprendere. Si è deciso quindi di dare 2 valori di incremento differenti al g_val: +10 per le azioni di sali/scendi dalla metropolitana e un +5 per le azioni di vai. Con questi pesi le azioni di sali/scendi risultano molto più costose dell'azione di vai, facendo prediligere quindi percorsi più lunghi con pochi cambi di

metro a percorsi più brevi ma con cambi più frequenti. Per quanto riguarda l'euristica abbiamo utilizzato la distanza euclidea. La distanza euclidea viene calcolata come segue:

$$L1(p1, p2) = sqrt((x1 - x2) + (y1 - y2))$$

Il risultato di questa semplice operazione sarà l'f_val associata ad ogni nodo nella ricerca.

2.2 Ricerca in profondità ad approfondimento iterativo con stima (IDA*)

2.2.1 Il codice sviluppato

```
1  ric_prof_lim(S,Depth,_,_,[]) :-
2          f_val(F),
3          F =< Depth,
4          finale(S),!.
5  ric_prof_lim(S,Depth,G,Visitati,[Az|Resto]) :-
6          f_val(F),
7          F =< Depth,
8          applicabile(Az,S),
9          trasforma(Az,S,Nuovo_S),
10         \+ member(Nuovo_S,Visitati),
11         num_nodi_open,
12         G1 is G + 1,
13         calcolo_euristica(Nuovo_S,G1),
14         ric_prof_lim(Nuovo_S,Depth,G1,[S|Visitati],Resto).
15 ric_prof_lim(_,Depth,_,_,_) :-
16         f_val(F),
17         F > Depth,
18         try_prof(F),
19         fail.
20
21 try_prof(F) :-
22         soglia(X),
23         X =< F, !
24         ;
25         retract(soglia(X)), !,
26         assert(soglia(F)).
27
28 num_nodi_open:-
29         nb_getval(counter, N1),
30         New1 is N1 + 1,
31         nb_setval(counter, New1).
32
33 ric_idastar(I,Ris,Depth,G) :- ric_prof_lim(I,Depth,G,[],Ris).
34
35 ric_idastar(I,Ris,_,G) :-
36         soglia(Sog),
37         retract(soglia(Sog)),
38         assert(soglia(99999)),
39         ric_idastar(I,Ris,Sog,G).
40
41 idastar :-
42         iniziale(S),
```

```
43          nb_setval(counter , 0),
44          calcolo_euristica(S,0),
45          f_val(D),
46          time(ric_idastar(S,Ris,D,0)),
47          nb_getval(counter, N_res),
48          writeln(Ris),
49          write(N_res),
50          write('\n').
```

2.2.2 Analisi dettagliata della strategia

L'implementazione dell'algoritmo è stata fatta in modo da sfruttare il lavoro già fatto con la ricerca in profondità con approfondimento iterativo, alla quale andiamo ad aggiungere un euristica specifica per ogni dominio che fornirà un valore di soglia per la nostra ricerca in profondità. La soglia viene inizializzata con un valore di default molto grande, ma man mano che l'esecuzione prosegue, viene sostituita di volta in volta con la f_val minima presa dai nodi che hanno superato la soglia nell'iterazione precedente. Più nel dettaglio l'algoritmo inizia il suo lavoro settando la f_val che verrà utilizzata come limite di ricerca e richiamando la ricerca in profondità già sviluppata precedentemente con alcune piccole modifiche. Nel caso base è stato aggiunto un controllo sulla profondità, ovvero che la nostra f_val deve essere minore o uguale alla profondità; se è cosi si procede con il semplice controllo di S come stato finale, altrimenti si passa alla regola per aggiornare la soglia. Per quanto riguarda il caso generico, abbiamo sempre il controllo della f_val sulla profondità e in più abbiamo il suo ricalcolo tramite l'euristica specifica del dominio. La regola try_prof(F) ha una doppia funzione: oltre ad aggiornare la soglia quando questa risulta maggiore della nostra f_val corrente, ci permette di bloccare la ricerca quando siamo nel caso opposto, ovvero che la f_val risulta maggiore del nostro valore di soglia fissato. Infine quando la ricerca in profondità fallisce abbiamo una regola per riportare al valore di default la soglia e ricominciare nuovamente la ricerca in profondità.

2.2.3 Dominio del Mondo dei Cammini

Per quanto riguarda l'esempio fornito dal Prof. Martelli (cammini 10x10) l'algoritmo precedentemente descritto, sfruttando l'euristica descritta al punto 2.1, è in grado di fornire la soluzione ottima dopo 0.123 secondi, visitando 13078 nodi e facendo 681842 inferenze. Anche sull'esempio del professor. Torasso (cammini 20x20) l'algoritmo trova la soluzione in 0.146 secondi, visitando 15796 nodi e facendo 791573 inferenze. Qui di seguito abbiamo i grafici riassuntivi dei risultati ottenuti.

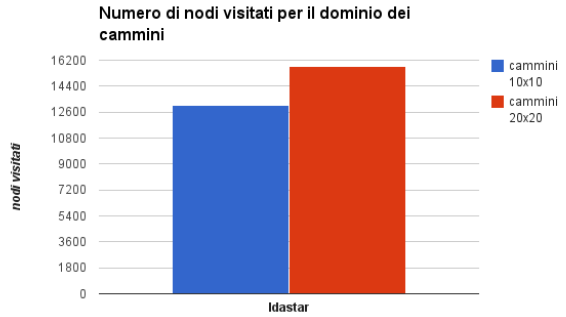

Figura 2.2: Nodi visitati per i domini dei cammini

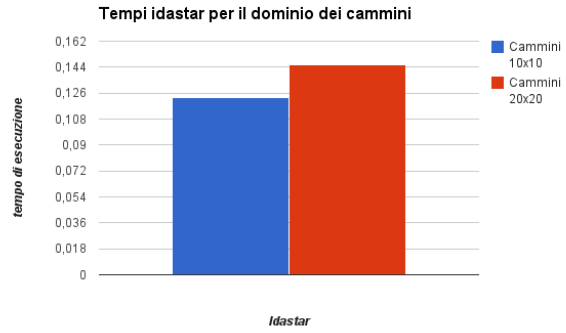

Figura 2.3: Tempi di calcolo per i domini dei cammini

2.2.4 Dominio del Mondo dei Blocchi

Analizziamo il dominio dei blocchi con le due euristiche precedentemente descritte. Come possiamo notare dalle tabelle e dai successivi grafici, l'euristica

	Eur. Locale (dom. 10x10)	Eur. Globale (dom. 10x10)	Eur. Locale (dom. 20x20)	Eur. Globale (dom. 20x20
Tempo	0.076	0.007	14.564	0.047
Inferenze	477443	46834	89033213	433972
Nodi Visitati	1444	71	183364	327

globale risulta essere migliore rispetto a quella locale.

Figura 2.4: Grafici per il dominio dei cammini.

2.3 Ricerca in ampiezza sui grafi con stima (A*)

2.3.1 Il codice sviluppato

```
1  ric_astar([nodo(_,_, S, Lista_Az)|_],_, Lista_Az) :- finale(S), !.
2  ric_astar([nodo(Fcost, Gcost, S, Lista_Az)| R_lista_open], Closed, Lista_Ris) :-
3          member(S, Closed) ->
4                  ric_astar(R_lista_open, Closed, Lista_Ris);
5          num_nodi_open,
6          open_node(nodo(Fcost, Gcost, S, Lista_Az), Lista_children),
7          ord_union(Lista_children, R_lista_open, Nuova_open),
8          ric_astar(Nuova_open,[S|Closed],Lista_Ris).
9
10 open_node(nodo(Fcost, Gcost, S, Lista_Az), Lista_childern) :-
11         findall(Az, applicabile(Az,S), Az_applicabili),
12         best_node(nodo(Fcost, Gcost, S, Lista_Az), Az_applicabili, Lista_childern).
13
14 best_node(_,[],[]).
15 best_node(nodo(Fcost, Gcost, S, Lista_Az), [Az|R_az], Lista_children) :-
16         trasforma(Az, S, Nuovo_S),
17         append(Lista_Az, [Az], Nuova_lista_az),
18         % num_nodi_open,
19         best_node(nodo(Fcost, Gcost, S, Lista_Az), R_az, Old_children),
20         G1 is Gcost + 1,
21         calcolo_euristica(Nuovo_S, G1),
22         f_val(F),
23         ord_add_element(Old_children, nodo(F, G1, Nuovo_S, Nuova_lista_az), Lista_children).
24
25 num_nodi_open:-
26         nb_getval(counter, N1),
27         New1 is N1 + 1,
28         nb_setval(counter, New1).
29
30 astar :-
31         iniziale(S),
32         nb_setval(counter , 0),
33         calcolo_euristica(S, 0),
34         f_val(Fcost),
35         time(ric_astar([nodo(Fcost, 0, S, [])], [], Ris)),
36         nb_getval(counter, N_res),
37         writeln(Ris),
38         write(N_res),
39         write('\n').
```

2.3.2 Analisi dettagliata della strategia

L'implementazione dell'algoritmo di ricerca astar è stato fatto in modo molto semplice. Sfruttando le regole native di Prolog ord_union() e ord_add_element possiamo costruire delle liste di nodi ordinate secondo il loro f_val() cosi da avere la certezza di analizzare sempre lo stato migliore fra quelli all'interno della lista degli stati da visitare. Vediamo più nel dettaglio l'implementazione. Come al solito la ricerca astar è composta di 2 regole principali, una per il caso base che non fa altro che controllare che lo stato in input S sia lo stato finale e una per caso generico. La regola per il caso generico controlla prima di tutto che lo stato in inpunt non faccia parte della lista

dei nodi chiusi e quindi già visitato; in caso affermativo, richiamiamo la ricerca astar sul nodo successivo nella lista dei nodi aperti, mentre in caso negativo lo apriamo con la regola open_node(). La regola non fa altro che cercare tutte le azioni applicabili dal nodo corrente, inserendole in una lista di azioni che verrà data in pasto alla regola best_node(). La regola best_node() non fa altro che costruire una lista ordinata di figli basata sul loro f_val() in ordine crescente. La costruzione viene fatta grazie a ord_add_element (Set1, Elem,SetRes) il quale prende in input una lista ordinata di elementi (Set1) alla quale aggiunge in modo ordinato il nuovo elemento (Elem) mettendo infine il risultato nella lista ordinata SetRes.Terminata la costruzione di questa lista ordinata di nodi figlio, la aggiungiamo alla lista dei nodi aperti tramite la regola ord_union(Set1,Set2,SetRes) la quale non fa altro che unire ordinatamente le due liste mettendo nuovamente il risultato in SetRes. Come ultimo passo richiamiamo la ricerca astar sul primo nodo della lista dei nodi aperti e aggiungiamo il nodo corrente alla lista dei nodi chiusi.

2.3.3 Dominio del Mondo dei Cammini

Per quanto riguarda l'esempio fornito dal Prof. Martelli (cammini 10x10) l'algoritmo precedentemente descritto, sfruttando l'euristica descritta al punto 2.1 impiega 0.029 secondi per trovare la soluzione; visita 53 nodi ed esegue 19517 inferenze per arrivare al goal. Anche con l'esempio del professor Torasso (cammini 20x20) l'algoritmo impiega 0.037 secondi, visitando 128 nodi e facendo 65057 inferenze per trovare la soluzione. Qui di seguito abbiamo i grafici riassuntivi dei risultati ottenuti.

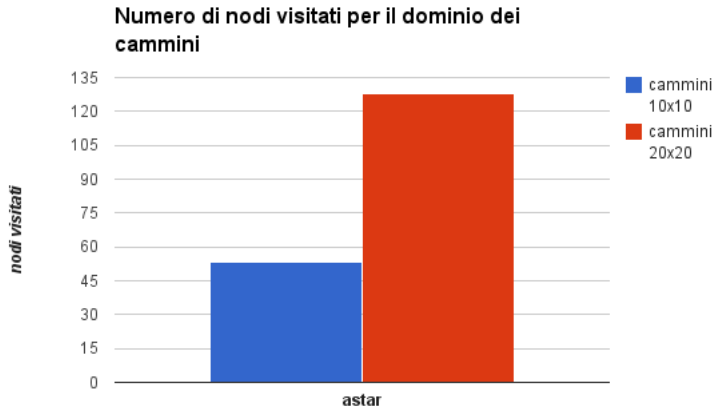

Figura 2.5: Nodi visitati per i domini dei cammini

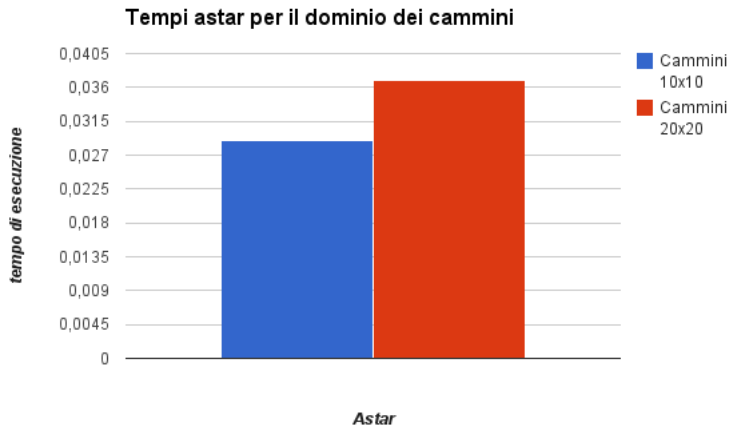

Figura 2.6: Tempi di calcolo per i domini dei cammini

2.3.4 Dominio del Mondo dei Blocchi

Analizziamo il dominio dei blocchi con le due euristiche precedentemente descritte. Qui di seguito abbiamo i grafici riassuntivi dei risultati ottenuti.

	Eur. Locale (dom. 10x10)	Eur. Globale (dom. 10x10)	Eur. Locale (dom. 20x20)	Eur. Globale (dom. 20x20
Tempo	0.072	0.036	N.P.	1.60
Inferenze	396947	191916	N.P.	949393
Nodi Visitati	304	99	N.P.	150

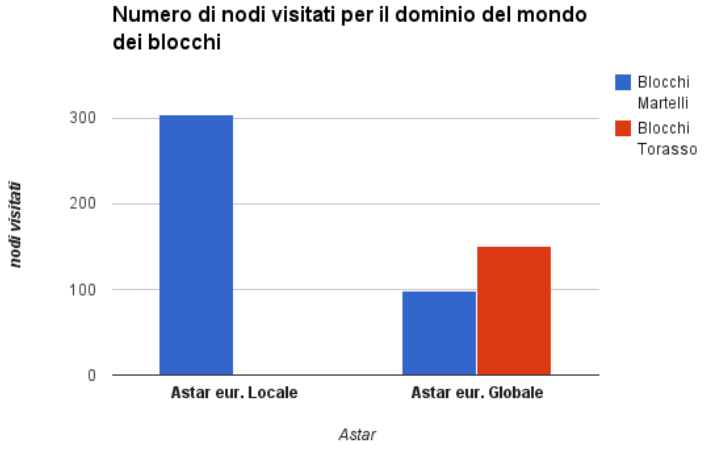

Figura 2.7: Nodi visitati per i domini del mondo dei blocchi

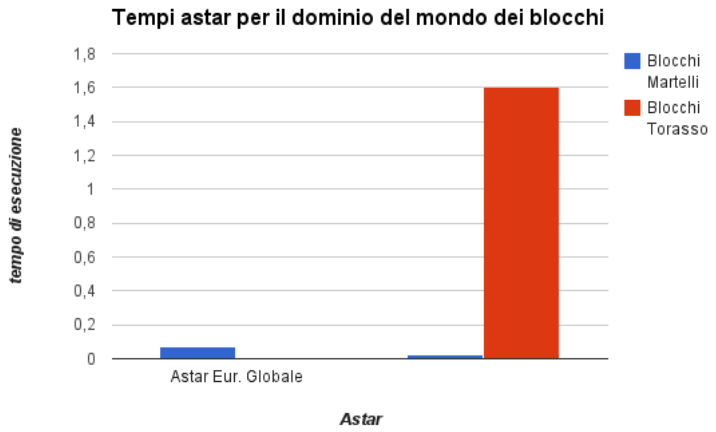

Figura 2.8: Tempi di calcolo per i domini del mondo dei blocchi

2.3.5 Dominio Metropolitana di Londra

Il codice sviluppato

```
1  calcolo_euristica([at(Stazione1),_],[at(Stazione2),_]) :-
2          stazione(Stazione1, R, C),
3          stazione(Stazione2, R1, C1),
4          X is (R - R1)^2,
5          Y is (C - C1)^2,
6          abs(X, Xabs),
7          abs(Y, Yabs),
8          H is Xabs + Yabs,
9          F is sqrt(H),
10         retract(f_val(_)),
11         assert(f_val(F)),
12         retract(h_val(_)),
13         assert(h_val(H)).
14
15 ric_astar([nodo(_,_, S, Lista_Az)|_],_, Lista_Az) :- finale(S), !.
16 ric_astar([nodo(Fcost, Gcost, S, Lista_Az)| R_lista_open], Closed, Lista_Ris) :-
17         member(S, Closed) ->
18                 ric_astar(R_lista_open, Closed, Lista_Ris);
19         num_nodi_open,
20         open_node(nodo(Fcost, Gcost, S, Lista_Az), Lista_children),
21         ord_union(Lista_children, R_lista_open, Nuova_open),
22         ric_astar(Nuova_open,[S|Closed], Lista_Ris).
23
24
25 open_node(nodo(Fcost, Gcost, S, Lista_Az), Lista_childern) :-
26         findall(Az, applicabile(Az,S), Az_applicabili),
27         best_node(nodo(Fcost, Gcost, S, Lista_Az), Az_applicabili, Lista_childern).
28
29 best_node(_,[],[]).
30 best_node(nodo(Fcost, Gcost, S, Lista_Az), [Az|R_az], Lista_children) :-
31         finale(Goal),
32         trasforma(Az, S, Nuovo_S),
33         append(Lista_Az, [Az], Nuova_lista_az),
34         % num_nodi_open,
35         best_node(nodo(Fcost, Gcost, S, Lista_Az), R_az, Old_children),
```

```
36          calcola_G(Az, Gcost, G1),
37          calcolo_euristica(Nuovo_S, Goal),
38          f_val(F),
39          ord_add_element(Old_children, nodo(F, G1, Nuovo_S, Nuova_lista_az), Lista_children).
40
41  num_nodi_open:-
42          nb_getval(counter, N1),
43          New1 is N1 + 1,
44          nb_setval(counter, New1).
45
46  calcola_G(sali(_,_), Gcost, G1) :-
47          G1 is Gcost + 10.
48
49  calcola_G(scendi(_), Gcost, G1) :-
50          G1 is Gcost + 10.
51
52  calcola_G(vai(_,_,_,_), Gcost, G1) :-
53          G1 is Gcost + 5.
54
55  astar :-
56          iniziale(S),
57          nb_setval(counter , 0),
58          time(ric_astar([nodo(0, 0, S, [])], [], Ris)),
59          nb_getval(counter, N_res),
60          writeln(Ris),
61          write(N_res),
62          write('\n').
```

2.3.6 Analisi dettagliata della strategia

L'algoritmo sviluppato calcola una soluzione in soli 0.126 secondi. Di particolare rilievo è il numero di nodi visitati per trovare la soluzione, che è di soli 10 nodi. Il numero coincide esattamente con i nodi della soluzione. Su un altro esempio generato manualmente (partenza da Holborn e arrivo a Waterloo), l'algoritmo trova la soluzione in 0.071 secondi ma questa volta il numero di nodi aperti è più alto del numero dei nodi della soluzione (15 nodi aperti contro i 9 della soluzione).

Parte II

Clingo

Capitolo 3

Answer Set Programming

In questo capitolo mostriamo le soluzioni realizzate per i problemi sui vincoli proposti. Per risolvere questi problemi abbiamo fatto uso dell Answer Set Solver *Clingo*.

3.1 Problema delle Cinque Case

In questa parte verrà descritto lo svolgimento dell'esercizio delle Cinque Case. In questo caso è stato utilizzata la versione 4.4.0 di clingo.

Il problema viene così enunciato:

> Cinque persone di nazionalità diverse vivono in cinque case allineate lungo una strada, esercitano cinque professioni distinte, e ciascuna persona ha un animale favorito e una bevanda favorita, tutti diversi fra loro. Le cinque case sono dipinte con colori diversi. Sono noti i seguenti fatti:
>
> 1. L'inglese vive nella casa rossa.
> 2. Lo spagnolo possiede un cane.
> 3. Il giapponese è un pittore.
> 4. L'italiano beve tè.
> 5. Il norvegese vive nella prima casa a sinistra.
> 6. Il proprietario della casa verde beve caffè.
> 7. La casa verde è immediatamente sulla destra di quella bianca.
> 8. Lo scultore alleva lumache.
> 9. Il diplomatico vive nella casa gialla.
> 10. Nella casa di mezzo si beve latte.
> 11. La casa del norvegese è adiacente a quella blu.
> 12. Il violinista beve succo di frutta.
> 13. La volpe è nella casa adiacente a quella del dottore.
> 14. Il cavallo è nella casa adiacente a quella del diplomatico.

L'obiettivo consiste nel trovare chi ha come animale domenstico la *giraffa*.

```
1  % houses
2  house_position(1..5).
3  house_col(red;green;blue;yellow;white).
4  house_per(eng;spa;jap;ita;nor).
5  house_prof(painter;sculptor;diplomatic;doctor;violinist;).
6  house_ani(dog;zebra;horse;fox;snail).
7  house_bev(coffee;milk;tea;juice;beer).
8
9  % (Directed) Edges
10 adj(1,2).
11 adj(2,3).
12 adj(3,4).
13 adj(4,5).
14
15 % Generate
16 1 { house(P,C,N,PR,A,B) : house_col(C), house_per(N), house_prof(PR), house_ani(A),
       house_bev(B)} 1 :- house_position(P).
17
18 % different colors
19 :- X != Y, house(X,C,_,_,_,_), house(Y,C,_,_,_,_).
20 % different nationality
21 :- X != Y, house(X,_,N,_,_,_), house(Y,_,N,_,_,_).
22 % different prof
23 :- X != Y, house(X,_,_,PR,_,_), house(Y,_,_,PR,_,_).
24 % different animals
25 :- X != Y, house(X,_,_,_,AN,_), house(Y,_,_,_,AN,_).
26 % different beverage
27 :- X != Y, house(X,_,_,_,_,B), house(Y,_,_,_,_,B).
28 % inglese rosso
29 :- house(_,red,P,_,_,_), P != eng.
30 % spagnolo possiede cane
31 :- house(_,_,P,_,dog,_), P != spa.
32 % giapponese pittore
33 :- house(_,_,P,painter,_,_), P != jap.
34 % italiano beve te
35 :- house(_,_,P,_,_,tea), P != ita.
36 % norvegese prima casa a sinistra
37 :- house(1,_,P,_,_,_), P != nor.
38 % proprietario casa verde beve caffe
39 :- house(_,C,_,_,_,coffee), C != green.
40 % casa verde destra bianca
41 :- house(X,white,_,_,_,_), house(Y,green,_,_,_,_), not adj(X,Y).
42 % scultore alleva lumache
43 :- house(_,_,_,P,snail,_), P != sculptor.
44 % diplomatico nella casa gialla
45 :- house(_,yellow,_,P,_,_), P != diplomatic.
46 % casa 3 si beve latte
47 :- house(3,_,_,_,_,B), B != milk.
48 % norv adiacente blue
49 :- house(2,C,_,_,_,_), C != blue.
50 % violinista succo di frutta
51 :- house(_,_,_,P,_,juice), P != violinist.
52 % volpe adiacente dottore
53 :- house(X,_,_,_,fox,_), house(Y,_,_,doctor,_,_), not adj(X,Y), not adj(Y,X).
54 % cavallo adiacente diplomatico
55 :- house(X,_,_,_,horse,_), house(Y,_,_,diplomatic,_,_), not adj(X,Y), not adj(Y,X).
```

Dalla linea 2 alla linea 7 (codice sottostante) sono stati dichiarati i fatti del problema: il numero delle case, i colori disponibili, le nazionalità, le professioni gli animali e le bevande. Le linee dalla 10 alla 13 servono per descrivere il concetto

di adiacenza delle case. La linea 16 serve per generare tutti i possibili modelli. Le linee dalla 19 alla 55 servono per descrivere i vincoli (*integrity constraint*) che il nostro modello dovrà soddisfare.

In particolare i vincoli per assegnare ogni casa consistono di:

1. color: non possono esserci case diverse con colore uguale (riga 19).
2. nationality: non possono esistere due case diverse il cui abitante ha la stessa nazionalità (riga 21).
3. profession: non possono esserci due case diverse con uguale professione (riga 23).
4. animal: stesso vincolo di sopra ma prendendo in considerazione gli animali (riga 27).
5. beverage: stesso vincolo di sopra ma prendendo in considerazione le bevande (riga 27).

Dalla riga 28 alla 55 vengono descritti i vincoli specifici dei fatti conosciuti.

Il concetto generale qui applicato consiste nel creare un solo predicato contenente tutti i termini che descrivono una singola casa. In particolare il predicato *house* è definito nel seguente modo:

```
1 house(id_position,
2         color,
3         resident_nationality,
4         resident_profession,
5         resident_pet,
6         resident_best_beverage)
```

I termini del predicato dovrebbero essere auto-esplicanti. Nel codice i vincolo sono espressi tramite l'uso della sintassi con underscore (es. $pred(N_{,})$), che equivale ad una wild-card.

Di seguito la soluzione trovata dal programma:

```
1 house(1,yellow,nor,diplomatic,fox,beer)
2 house(2,blue,ita,doctor,horse,tea)
3 house(3,red,eng,sculptor,snail,milk)
4 house(4,white,spa,violinist,dog,juice)
5 house(5,green,jap,painter,zebra,coffee)
```

3.2 Pianificazione tramite Answer Set Programming

Un approccio alla pianificazione è basato sulla verifica di soddisfacibilità di una formula proposizionale, che rappresenta il problema di pianificazione. In ASP il problema viene modellato tramite regole della programmazione proposizionale, che ammette la negazione per fallimento. Questo approccio (oltre a SATPLAN) ha una affinità con i grafi di pianificazione descritti da *GRAPHPLAN*.

Un grafo di pianificazione in *GRAPHPLAN* è un grafo orientato organizzato in livelli: $S_0, A_0, S_1, A_1, ..., A_2, S_{n+1}$.

Il livello S_i è l'insieme dei nodi che rappresentano fluenti validi in S_i. Il livello A_i è l'insieme dei nodi che rappresentano le azioni applicabili in S_i. Ci sarà quindi

un'alternanza di livelli S_i e A_i, fino alla condizione di terminazione. Lo stato S_{n+1} rappresenta lo stato che contiene il goal.

Per rappresentare il problema di pianificazione attraverso *GRAPHPLAN* abbiamo bisogno di due predicati che, rispettivamente, rappresentino i fluenti (sia veri che falsi) e le azioni eseguite in ogni stato.

1. *holds(F,S)*: in cui F è un fluente ed S è il livello S_i, in F è valido.
2. *occurs(A,L)*: che indica che una action A è eseguito nel livello L (in questo il livello L è un A_i).

In clingo i fluenti enumerabili sono stati formulati attraverso il predicato *fluent(F) :- proposition(term)*, mentre le azioni possibili nel dominio sono state formulate attraverso il predicato *action(A) :- proposition(term)*.

Abbiamo inoltre introdotto un costrutto sintattico per generare tutte le possibili *occurs(A,L)* e permettere più azioni possibili contemporaneamente.

```
1{occurs(A,L): action(A)} :- level(L).
```

Gli *Effetti* delle *occurs(A,L)*, ossia delle azioni, sono stati descritti tramite la seguente regola che esprime l'aggiunta di un fluente allo stato successivo.

```
holds(F,S+1) :- occurs(A,S), state(S)
```

I mutex sono stati espressi nella forma di una *regola false*, ossia:

```
:- occurs(A,S), holds(F1,S), holds(F2,S), ..., holds(FN,S), state(S)
```

Per rappresentare le regole di persistenza (dette anche no-op), possiamo utilizzare delle regole non monotone espresse nel modo seguente:

```
holds(F, S+1) :-
  fluent(F), state(S),
  holds(F,S), not -holds(F,S+1).
```

Queste regole "copiano" un fluente dallo stato S allo stato S+1 se e solo se nello stato S è presente il fluente F ma non la sua negazione nello stato S+1.

La regola di persistenza è usata anche per rendere persistenti i predicati -*holds(F,S)*, che esprimono una negazione forte.

Il goal viene formulato tramite il vincolo:

```
goal :- descrizione stato finale
:- not goal
```

Poiché dobbiamo mantenere gli stati completi, ossia mantengono F o $\neg F$ per ogni fluent F, dobbiamo aggiungere le regole causali. Queste regole asseriscono i predicati *-holdsF,S* se nello stato S non è presente il fluente F o se non è possibile dedurlo.

3.3 Problema dell'Air Cargo

Il codice sottostante illustra come il problema di pianificazione air cargo, descritto nel paragrafo 10.1.1 del *Russel-Norvig*, possa essere implementato mediante l'utilizzo della ASP.

Nella *prima parte* del codice (righe 1-4) troviamo la generazione dei predicati che descrivno gli stati e i livelli.

Nella *seconda parte* troviamo, le *azioni eseguibili* e i *fluenti ammessi* nel dominio:

1. le azioni eseguibili (8,9,10)
   (a) load(C,P,A): carica un cargo C all'interno dell'aereo P da un aeroporto A.
   (b) unload(C,P,A): scarica una cargo C dall'aereo P in un aeroporto (A).
   (c) fly(P,A1,A2): sposta l'aereo P dall'aeroporto A1 all'aeroporto A2.
2. la generazione delle *occurs(F,S)*, che consente più azioni contemporaneamente (riga 13)
3. i fluenti esistenti
   (a) in(C,P): il cargo C è caricato nell'aereo P.
   (b) at(P,A): il plane P è nell'aeroporto A.
   (c) at(C,A): il cargo C è nell'aeroporto A.

Nella *terza parte* definiamo gli effetti delle azioni, le precondizioni, le regole di persistenza e le regole causali:

1. gli effetti delle azioni (28-32)
   (a) load: nello stato $S+1$, il cargo C è nell'aereo P se viene effettuata una azione di load.
   (b) unload: nello stato $S+1$, il cargo C è nell'aeroporto A se viene effettuata una azione di unload.
   (c) fly: l'aereo è in aeroporto A2 se viene effettuata una azione di fly.
2. precondizioni delle azioni load, unload e fly:
   (a) load: non è possibile applicare l'azione di load se il cargo C si trova in qualche altro aereo P_1 o se il cargo C e l'aereo P non si trovano nello stesso aeroporto A.
   (b) unload: non è possibile applicare l'azione di unload se il cargo C non è all'interno dell'aereo P e se l'aereo P non si trova nell'aeroporto A.
   (c) fly: non è possibile applicare l'azione di fly se l'aereo P non si trova nell'aeroporto $A1$
3. regole di persistenza: svolgono il compito delle azioni di *no-op*, già spiegate nella parte introduttiva.
4. regole causali

Nella *quarta e ultima parte* definiamo lo stato iniziale, la descrizione del goal e le istruzioni di post-processing per specifiche di clingo.

Qui di seguito il codice completo:

```
1  #const lastlev=300.
2
3  level(0..lastlev).
4  state(0..lastlev+1).
5
6  % AZIONI
7
8  action(load(C,P,A)) :- cargo(C), plane(P), airport(A).
9  action(unload(C,P,A)) :- cargo(C), plane(P), airport(A).
10 action(fly(P,A1,A2)) :- plane(P), airport(A1), airport(A2), A1 != A2.
11
12 % le azioni possono essere eseguite in parallelo
13 1{occurs(A,L): action(A)} :- level(L).
14
15 % FLUENTI
16
17 % cargo C is in plane P
18 fluent(in(C,P)) :- cargo(C), plane(P).
19 % cargo C is at airport A
20 fluent(at(C,A)) :- cargo(C), airport(A).
21 % plane P is at airport A
22 fluent(at(P,A)) :- plane(P), airport(A).
23
24 % EFFETTI
25 % i -holds vengono generati dalle regole causali
26
27 % afferma che un cargo e' su un aereo, quando viene caricato
28 holds(in(C,P),S+1) :- occurs(load(C,P,A),S),state(S).
29 % rimette i cargo in un areoporto quando vengono scaricati
30 holds(at(C,A),S+1) :- occurs(unload(C,P,A),S),state(S).
31 % mette l'aereo nel nuovo areoporto quando l'azione fly deve essere fatta
32 holds(at(P,AD),S+1) :- occurs(fly(P,AS,AD),S),state(S).
33
34 % PRECONDIZIONI
35
36 % precondizioni di laod
37 % - il cargo non deve essere su un aereo
38 % - il cargo e l'aereo devono essere nello stesso areoporto
39 :- occurs(load(C,P0,A),S), holds(in(C,P1),S).
40 :- occurs(load(C,P,A),S), -holds(at(C,A),S).
41 :- occurs(load(C,P,A),S), -holds(at(P,A),S).
42
43 % precondizioni di unload
44 % - il cargo deve essere su quell'aereo, l'aereo in quell'areoporto
45 :- occurs(unload(C,P,A),S), -holds(in(C,P),S).
46 :- occurs(unload(C,P,A),S), -holds(at(P,A),S).
47 % - il cargo non deve essere in un areoporto
48 :- occurs(unload(C,P,A),S), holds(in(C,A),S).
49
50 % precondizioni di fly
51 % - il l'aereo deve essere in A1 e non essere in A2
52 :- occurs(fly(P,A1,A2),S), -holds(at(P,A1),S).
53
54 % trick
55 % lo stesso aereo non puo' eseguire piu' azioni nello stesso livello
56 :- occurs(fly(P,A1,A2),S), occurs(load(C,P,A1),S), plane(P), airport(A1), airport(A2), cargo(C).
57 :- occurs(fly(P,A1,A2),S), occurs(unload(C,P,A1),S), plane(P), airport(A1), airport(A2), cargo(C).
58
59 % PERSISTENZA
60 holds(F, S+1) :-
61   fluent(F), state(S),
```

```
62    holds(F,S), not −holds(F,S+1).
63
64  −holds(F,S+1) :−
65    fluent(F), state(S),
66    −holds(F, S), not holds(F, S+1).
67
68  % REGOLE CAUSALI
69
70  % cargo non e' in nessun aereo, se e' in areoporto
71  −holds(in(C,P),S) :− cargo(C), airport(A), plane(P), state(S), holds(at(C,A),S).
72  % cargo e' in plane, cargo non e' in nessun areoporto
73  −holds(at(C,A),S) :− cargo(C), airport(A), plane(P), state(S), holds(in(C,P),S).
74  % cargo e' in areoporto, non e' in tutti gli altri
75  −holds(at(C,A2),S) :− cargo(C), airport(A1), airport(A2), state(S), holds(at(C,A1),S), A1!=A2.
76  % aereo in areoporto, non e' in tutti gli altri
77  −holds(at(P,A2),S) :− plane(P), airport(A1), airport(A2), state(S), holds(at(P,A1),S), A1!=A2.
78
79  :− not goal.
80
81  #hide.
82  #show occurs/2.
```

3.3.1 Problema 1

Questo primo problema di pianificazione *air cargo* è lo stesso presente nel paragrafo 10.1.1 del *Russel&Norvig*

Lo stato iniziale e il goal del problema sono:

```
1  cargo(c1).
2  cargo(c2).
3  plane(p1).
4  plane(p2).
5  airport(jfk).
6  airport(sfo).
7
8  holds(at(c1,sfo),0).
9  holds(at(c2,jfk),0).
10 holds(at(p1,sfo),0).
11 holds(at(p2,jfk),0).
12
13 goal:− holds(at(c1,jfk),lastlev+1),
14         holds(at(c2,sfo),lastlev+1).
```

Una soluzione ottima, ottenuta assegnando a *lastlevel* il valore 2 é:

```
1  occurs(load(c1,p1,sfo),0)
2  occurs(load(c2,p2,jfk),0)
3  occurs(fly(p1,sfo,jfk),1)
4  occurs(fly(p2,jfk,sfo),1)
5  occurs(unload(c1,p1,jfk),2)
6  occurs(unload(c2,p2,sfo),2)
```

il tempo impiegato è *infinitesimo*.

3.3.2 Problema 2

Il secondo problema è stato formulato in modo da ampliare il numero di cargo da spostare.

```
cargo(c1).
cargo(c2).
cargo(c3).
cargo(c4).
cargo(c5).
plane(p1).
plane(p2).
airport(jfk).
airport(sfo).
airport(bari).

holds(at(p1,sfo),0).
holds(at(p2,jfk),0).

holds(at(c1,sfo),0).
holds(at(c2,sfo),0).
holds(at(c3,sfo),0).
holds(at(c4,sfo),0).
holds(at(c5,sfo),0).

% GOAL
goal:- holds(at(p1,sfo), lastlev+1),
       holds(at(p2,jfk), lastlev+1),
       holds(at(c1,bari),lastlev+1),
       holds(at(c2,bari),lastlev+1),
       holds(at(c3,bari),lastlev+1),
       holds(at(c4,bari),lastlev+1),
       holds(at(c5,bari),lastlev+1).
```

Una delle soluzioni ottenute, impostando *lastlev* a 3 è la seguente.

```
occurs(load(c5,p1,sfo),0)
occurs(load(c4,p1,sfo),0)
occurs(load(c3,p1,sfo),0)
occurs(load(c2,p1,sfo),0)
occurs(load(c1,p1,sfo),0)
occurs(fly(p2,jfk,bari),0)
occurs(fly(p2,bari,jfk),1)
occurs(fly(p1,sfo,bari),1)
occurs(unload(c5,p1,bari),2)
occurs(unload(c4,p1,bari),2)
occurs(unload(c3,p1,bari),2)
occurs(unload(c2,p1,bari),2)
occurs(unload(c1,p1,bari),2)
occurs(fly(p2,jfk,bari),2)
occurs(fly(p2,bari,jfk),3)
occurs(fly(p1,bari,sfo),3)
```

Come si può notare, dato che non abbiamo inserito un vincolo di carico del plane, l'aereo $P1$ carica e scarica tutti i cargo in uno solo level, il secondo. L'aereo $P2$, inoltre, fa voli a vuoto, non essendo stata specificata nessuna regola di *viaggia a vuoto solo se c'è da caricare un cargo nell'altro areoporto.*

Il tempo, aumentando di un solo livello il grafo, è aumentato di 10millisecondi.

Usando lo stesso esempio, se aumentassimo il livello massimo (*lastlev*) a 30, invece che a 3, il tempo di esecuzione per trovare almeno una soluzione è di 110millisecondi, ovvero un aumento di 10 volte. Possiamo stimare quindi che il tempo di esecuzione è pari ad $O(n)$, dove n è il numero di livelli.

Appendice A

Code Repository

Ai fini di un adeguato meccanismo di versioning per la collaborazione tra gli studenti autori del progetto, si è scelto di utilizzare git come meccanismo di versioning. Questo ha comportato un sistema affidabile per il tracciamento di versioni e modifiche, con la possibilità di tenere traccia, ed eventualmente facendo delle operazioni di revert, degli sviluppi incrementali dei vari progetti. È stato reso disponibile il repository su github [1].

A.1 Il codice dei test

Qui di seguito si trova il codice specifico per i vari test eseguiti sui domini implementati

A.1.1 Cammini 10x10

```
1  occupata(pos(2,5)).
2  occupata(pos(3,5)).
3  occupata(pos(4,5)).
4  occupata(pos(5,5)).
5  occupata(pos(6,5)).
6  occupata(pos(7,5)).
7  occupata(pos(7,1)).
8  occupata(pos(7,2)).
9  occupata(pos(7,3)).
10 occupata(pos(7,4)).
11 occupata(pos(5,7)).
12 occupata(pos(6,7)).
13 occupata(pos(7,7)).
14 occupata(pos(8,7)).
15 occupata(pos(4,7)).
16 occupata(pos(4,8)).
17 occupata(pos(4,9)).
18 occupata(pos(4,10)).
19
20 iniziale(pos(4,2)).
```

[1] il repository è disponibile su github all'indirizzo https://github.com/andr3a87/NG_Cafe

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

A.1.2 Cammini 20x20

```
1  occupata(pos(7,15)).
2  occupata(pos(8,15)).
3  occupata(pos(9,15)).
4  occupata(pos(10,15)).
5  occupata(pos(11,15)).
6  occupata(pos(12,15)).
7  occupata(pos(13,15)).
8
9  occupata(pos(13,6)).
10 occupata(pos(13,7)).
11 occupata(pos(13,8)).
12 occupata(pos(13,9)).
13 occupata(pos(13,10)).
14 occupata(pos(13,11)).
15 occupata(pos(13,12)).
16 occupata(pos(13,13)).
17 occupata(pos(13,14)).
18
19 occupata(pos(15,1)).
20 occupata(pos(15,2)).
21 occupata(pos(15,3)).
22 occupata(pos(15,4)).
23 occupata(pos(15,5)).
24 occupata(pos(15,6)).
25 occupata(pos(15,7)).
26 occupata(pos(15,8)).
27 occupata(pos(15,9)).
28
29
30 iniziale(pos(10,10)).
31
32 finale(pos(20,20)).
```

A.1.3 Mondo dei blocchi del professor Martelli

```
1  block(a).
2  block(b).
3  block(c).
4  block(d).
5  block(e).
6
7  iniziale(S):-
8    list_to_ord_set([on(a,b),on(b,c),ontable(c),clear(a),on(d,e), ontable(e),clear(d),handempty],S).
9
10 goal(G):- list_to_ord_set([on(a,b),on(b,c),on(c,d),ontable(d), ontable(e)],G).
11
12 finale(S):- goal(G), ord_subset(G,S).
```

A.1.4 Mondo dei blocchi del professor Torasso

```
block(a).
block(b).
block(c).
block(d).
block(e).
block(f).
block(g).
block(h).


iniziale(S):-
        list_to_ord_set([clear(a), clear(c), clear(d), clear(e), clear(f), clear(g), clear(h), on(a,b),
        ontable(b), ontable(c), ontable(d), ontable(e), ontable(f), ontable(g), ontable(h),
            handempty],S).

goal(G):- list_to_ord_set([on(a,b),on(b,c),on(c,d),on(d,e),
        ontable(e)],G).
```

A.1.5 La metropolitana di Londra

Esempio 1:

```
stazione('Baker Street',4.5,5.6).
stazione('Bank',12,4).
stazione('Bayswater',1,3.7).
stazione('Bond Street',5.4,4.1).
stazione('Covent Garden',8,4).
stazione('Earls Court',0,0).
stazione('Embankment',8.2,3).
stazione('Euston',7.1,6.6).
stazione('Gloucester Road',1.6,0.6).
stazione('Green Park',6,2.8).
stazione('Holborn',8.6,4.8).
stazione('Kings Cross',8.2,7.1).
stazione('Leicester Square',7.6,3.6).
stazione('London Bridge',0,0).
stazione('Notting Hill Gate',0,3.2).
stazione('Oxford Circus',6.2,4.3).
stazione('Paddington',2.4,4.2).
stazione('Piccadilly Circus',7,3.3).
stazione('South Kensington',2.6,0.5).
stazione('Tottenham Court Road',7.4,4.5).
stazione('Victoria',5.8,1).
stazione('Warren Street',6.5,6).
stazione('Waterloo',9.2,2.4).
stazione('Westminster',8,1.8).

iniziale([at('Bayswater'),ground]).

finale([at('Covent Garden'),ground]).
```

Esempio 2:

```
iniziale([at('Holborn'),ground]).

finale([at('Waterloo'),ground]).
```