
INTELLIGENZA ARTIFICIALE E LABORATORIO:

PROLOG

27 settembre 2017

Andrea Forgione: 809240
Arthur Capozzi: 802586
Leonardo Zanchi: 800935

Indice

1 Domini di applicazione	3
1.1 Metropolitana di Londra	3
1.2 Mondo dei blocchi	4
2 Strategie di ricerca	6
2.1 Iterative deepening	6
2.1.1 Metropolitana di Londra	6
2.1.2 Mondo dei blocchi	8
2.2 A*	9
2.2.1 Metropolitana di Londra	9
2.2.1.1 Euristica della metropolitana di Londra	12
2.2.2 Mondo dei blocchi	12
2.2.2.1 Euristica del mondo dei blocchi	13
2.3 IDA*	14
2.3.1 Metropolitana di Londra	14
2.3.2 Mondo dei blocchi	17
3 Risultati	18
3.1 Metropolitana di Londra	18
3.1.1 Iterative deepening	18
3.1.2 A*	18
3.1.3 IDA*	19
3.2 Mondo dei blocchi	19
3.2.1 Iterative deepening	19
3.2.2 A*	19
3.2.3 IDA*	19

Introduzione

Questa è la relazione riguardante l'implementazione in prolog di alcune strategie di ricerca. La relazione è strutturata in tre parti: nella prima verranno elencati i due domini di applicazione, nella seconda saranno presentate le strategie, mentre nella terza risultati che queste hanno portato.

Capitolo 1

Domini di applicazione

I domini scelti tra quelli proposti in questa relazione sono due: la metropolitana di Londra e il mondo dei blocchi. In questo capitolo se ne darà una descrizione così che il lettore possa farsi un'idea di quella che fosse la richiesta.

1.1 METROPOLITANA DI LONDRA

Il primo dominio consiste in una rappresentazione ridotta del funzionamento della metropolitana della capitale inglese. Ridotta perché non tutte le linee sono prese in considerazione e non tutte le fermate presenti sono riportate. La richiesta in questo dominio è trovare i cammini che portino ad una certa stazione della metropolitana a partire da un'altra fermata di questa.

La descrizione formale del dominio è contenuta nel file *tube06-07.pl*. Sono possibili solo tre azioni: *sali/2*, *scendi/1* e *vai/4*. L'azione *sali/2* ha arità due e necessità della *Linea* e della *Direzione*. L'azione *scendi/1* ha arità uno e necessità della sola *Stazione*. L'azione *vai/4* ha arità e necessità della *Linea*, della *Direzione*, della *Stazione di partenza* e della *Stazione di arrivo*. La possibilità di svolgere un'azione è espressa dalla regola *applicabile*, mentre la reale applicazione dell'azione è svolta dalla regola *trasforma*.

Il fatto *percorso/3* rappresenta la nozione di linea; ha arità tre e viene specificato il nome della *Linea*, la *Direzione* ed infine una lista contentente la *Lista delle fermate*. Attraverso un regola *percorso*, invece, sono specificate le direzioni inverse delle fermate della metropolitana. La regola *tratta/4* consente di trovare la fermata successiva di una certa stazione.

Una serie di fatti *stazione/3* rappresentano le stazioni presenti nel modello metropolitana qui riportato; ha arità tre e specifica il nome della *Stazione* e la sua posizione. La regola *fermata/2*



Figura 1.1

rappresenta la stazione all'interno del percorso. Due fatti *iniziale/1* e *finale/1* specificano la stazione di partenza e la stazione di arrivo ricercata. Entrambi hanno arità uno.

1.2 MONDO DEI BLOCCHI

Il secondo dominio scelto è quello del mondo dei blocchi. È un dominio ampiamente citato in letteratura in cui un numero variabile di blocchi si trovano disposti su di un tavolo. Possono trovarsi direttamente poggiati sul tavolo, oppure sopra ad un altro blocco. Un blocco può essere spostato solo se non ha su di sé altri blocchi. I blocchi devono essere spostati fino ad arrivare alla configurazione finale voluta. La descrizione formale del problema si trova all'interno del file *blocchiord12.pl*.

Sono possibili quattro azioni differenti: *pickup/1*, *putdown/1*, *stack/1* e *unstack/1*. Tutte le

azioni hanno arità uno, dove viene specificato il blocco su cui compiere l'azione. Anche in questo dominio sono presenti due regole: *applicabile/2* e *trasforma/3*. Nel primo caso viene specificata l'azione da fare e lo stato in cui viene compiuta tale azione; nel secondo caso, invece, si specifica l'azione da realizzare, lo stato di partenza e quello di arrivo. L'azione *applicabile* viene utilizzata per verificare la possibilità o meno di applicare l'azione scelta, mentre *trasforma* realizza l'azione portando il sistema nel nuovo stato.

Una serie di fatti *block/1* rappresentano l'entità del blocco all'interno del nostro modello. Infine, tre fatti specificano lo stato iniziale del sistema, indicato con *iniziale/1* e lo stato finale, con *finale/1*.

Per dare un'idea del funzionamento della regola *applicabile* viene riportata quella che fa riferimento all'azione *unstack/2*.

```
applicable(unstack(X,Y),S) :-  
    block(X), block(Y), X\=Y,  
    ord_memberchk(on(X,Y),S),  
    ord_memberchk(clear(X),S),  
    ord_memberchk(handempty,S).
```

In questo caso la regola *unstack* sarà soddisfatta, e quindi l'azione *unstack/2* applicabile, quando esiste il blocco *X*, esiste il blocco *Y*, i blocchi sono diversi tra di loro, il blocc *X* si trova su *Y* nello stato *S*, il blocco *X* è libero (non ha blocchi sopra di lui) nello stato *S*, e la mano è libera nello stato *S*.

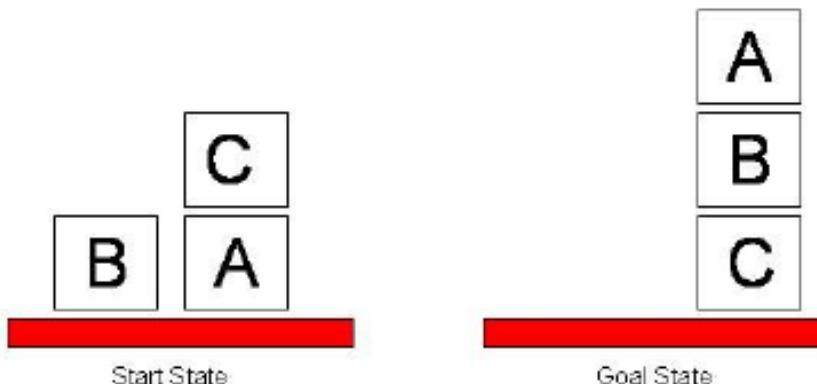


Figura 1.2

Capitolo 2

Strategie di ricerca

In questo capitolo vengono riportate le tre strategie di ricerca richieste e i risultati ottenuti applicandole ai domini riportati nel capitolo 1. Il capitolo è diviso in tre parti, una per ogni algoritmo implementato, e in ognuna di esse è descritto il funzionamento sui domini considerati.

2.1 ITERATIVE DEEPENING

È l'unica strategia di ricerca non informata presa in esame in questa relazione. L'idea chiave di *Iterative Deepening* (da ora *IDS*) è quella di ri-calcolare gli elementi della frontiera piuttosto che salvarli ogni volta. Viene utilizzata una ricerca in profondità limitata (*Depth-bounded depth-first search*, che viene incrementata ad ogni nuova ricerca se non si giunge allo stato goal. *IDS* combina i vantaggi della ricerca in ampiezza (*Breadth-first*) e quelli della ricerca in profondità(*Depth-first*), quindi ha i requisiti modesti di memoria della *Depth-first* e la completezza della *Breadth-first* quando il *branching factor* è finito. Inoltre, è una strategia di ricerca ottima quando il costo del cammino è una funzione monotona della profondità.

2.1.1 Metropolitana di Londra

In questa sezione viene riportato l'algoritmo *Iterative Deepening* applicato al dominio della metro di Londra. Sono allegati alla relazione i relativi sorgenti, debitamente commentati, ma se ne darà anche qua una descrizione dettagliata. Riguardo il binomio *IDS-metro* si fa riferimento ai file:

- *l_iterative_deepening_tube.pl*: è l'algoritmo che implementa la strategia di ricerca *Iterative Deepening*;

- *action_tube.pl*: in esso è descritto il dominio, con regole e fatti.

La regola *iterative_deepening/0* è il punto di entrata del programma e nel corpo ha tre goal che devono essere soddisfatti affinché la regola dia come risultato *true*: che esista il fatto *iniziale(StartNode* e che sia vero *iterative_deepening_search/3*. Il fatto finale semplicemente stampa la soluzione trovata. *iterative_deepening_search/3* è la regola che gestisce la chiamata alla ricerca in profondità limitata. Ne esistono due versioni, la prima (indicata nel listato con il commento %1) è la regola che richiama la ricerca in profondità limitata, mentre la seconda (indicata nel listato con il commento %2) si preoccupa di aggiornare la profondità di ricerca nel caso in cui l'iterazione precedente di ricerca non sia andata a buon fine. La prima, infatti, ha come parte del corpo la chiamata a *dfs_bound/4*, cioè la ricerca in profondità limitata. La seconda, invece, ha nel corpo l'incremento della profondità e la chiamata alla prima (*iterative_deepening_search/3*) con la nuova profondità. La regola *dfs_bound/4* (indicata con %3) è il cuore dell'implementazione e necessita del nodo attuale, della profondità di ricerca, di una lista dei nodi visitati ed infine una lista di azioni possibili su quel nodo. Nel corpo ci si accerta che:

1. la profondità sia maggiore di zero;
2. l'azione selezionata (quella in testa alla lista *[Action | OtherAction]*) sia applicabile al nodo attuale;
3. l'esecuzione dell'azione porti in un nuovo nodo;
4. il nuovo nodo trovato non appartenga alla lista dei nodi visitati (il comando *member/2* deve restituire *fail* come risultato, il quale verrà negato da
+);
5. si aggiorna la profondità riducendola di 1;
6. si chiama di nuovo la ricerca con la nuova profondità a partire dal nuovo nodo, aggiornando la lista nei nodi visitati.

La regola *dfs_bound/4* (indicata con %4) ha nel corpo una sola condizione per essere vera, cioè che si sia trovato lo stato finale della ricerca. Da notare l'utilizzo della *CUT*, che si rende necessaria per evitare che Prolog faccia *backtracking* e vada ad inficiare la soluzione trovata.

```

:- use_module(action_tube).

dfs_bounded(Node, _, _, []) :- %4
    finale(Node), !.

dfs_bounded(Node, MaxDepth, Visited, [Action | OtherAction]) :- %3
    MaxDepth > 0,
    applicabile(Action, Node),
    trasforma(Action, Node, NewNode),
    \+ member(NewNode, Visited),
    NewMaxDepth is MaxDepth - 1,
    dfs_bounded(NewNode, NewMaxDepth, [Node | Visited], OtherAction).

iterative_deepening_search(StartNode, MaxDepth, PathSolution) :- %1
    dfs_bounded(StartNode, MaxDepth, [], PathSolution).

iterative_deepening_search(StartNode, MaxDepth, PathSolution) :- %2
    NewMaxDepth is MaxDepth + 1,
    iterative_deepening_search(StartNode, NewMaxDepth, PathSolution).

iterative_deepening :-
    iniziale(StartNode),
    iterative_deepening_search(StartNode, 1, PathSolution),
    write(PathSolution).

```

2.1.2 Mondo dei blocchi

In questa sezione viene riportata l'implementazione della strategia di ricerca *Iterative deepening* applicata al dominio del mondo dei blocchi. In realtà, però, dato la modularità con cui è stato scritto il programma, la parte dell'algoritmo, contenuta nel file *1_iterative_deepening_block.pl*, è identica a quella già vista nella sezione *Iterative deepening* dedicata alla metropolitana di Londra. Per questo motivo non si riporterà in questa parte il modo in cui viene attuata la strategia. Altri due moduli fanno parte di questa soluzione: *configuration_block* e *action_block*. Il primo contiene i fatti che definiscono i blocchi, lo stato

iniziale e lo stato finale. In aggiunta, due configurazioni sono state realizzate, con numero diverso di blocchi e di conseguenza una complessità differente. Il secondo, invece, contiene la descrizione del dominio riportata nel capitolo 1, con *applicabile* e *transforma*.

2.2 A*

È un algoritmo di ricerca informata che fa uso di una euristica per stimare la distanza dal nodo attuale al nodo finale. A^* compie una ricerca nello spazio degli stati andando ad espandere ad ogn passo il nodo appartenente alla frontiera con costo $f(n)$ minore. La funzione $f(n)$ è definita come $f(n) = g(n) + h(n)$, dove $g(n)$ è il costo effettivo del cammino fino al nodo n , e $h(n)$ è una stima del costo del cammino dal nodo n al nodo finale. $f(n)$ può essere quindi definito come il costo stimato della soluzione più conveniente che passa per il nodo n .

La ricerca mediante l'algoritmo A^* trova la soluzione cercata, ma solo se l'euristica soddisfa certe condizioni:

- ammissibilità, cioè $h(n)$ non deve mai stimare per eccesso il costo per arrivare al nodo.
- consistenza, cioè la funzione euristica deve essere monotona, $h(n) \leq c(n, a, n') + h(n')$.

Le proprietà seguenti garantiscono una ricerca ottima da parte di A^* . In realtà, la consistenza è necessaria solo se la ricerca è effettuata su un grafo, mentre su un albero l'ammissibilità dell'euristica è sufficiente all'ottimalità.

2.2.1 Metropolitana di Londra

In questa sezione viene riportata l'implementazione dell'algoritmo A^* per il dominio della metropolitana di Londra. Sono allegati alla relazione i relativi sorgenti, che sono qui riportati:

- *3_astar_tube.pl* contiene l'implementazione dell'algoritmo di ricerca;
- *heuristic_tube.pl* contiene le euristiche utilizzate per la ricerca;
- *action_tube.pl* contiene la descrizione del dominio con regole e fatti;

In questo caso, quindi, il programma è stato diviso in tre moduli differenti, così da avere una maggiore versatilità, permettendo di adattare velocemente ad un altro dominio la stessa strategia di ricerca.

```
: - use_module(action_tube).
: - use_module(heuristic_tube).

astar_search([nodo(_, _, Node, ActionsAvailable) | _], _, ActionsAvailable
) :- %1
    finale(Node).

astar_search([nodo(F, G, Node, ActionsAvailable) | OtherActions],
    CloseNodes, PathSolution) :- %2
    member(Node, CloseNodes) -> astar_search(OtherActions, CloseNodes,
        PathSolution);
    expand(nodo(F, G, Node, ActionsAvailable), NextNodes),
    ord_union(OtherActions, NextNodes, Tail),
    astar_search(Tail, [Node | CloseNodes], PathSolution).

expand(nodo(F, G, Node, ActionsAvailable), NextNodes) :-
    findall(Action, applicabile(Action, Node), ListActions),
    successor(nodo(F, G, Node, ActionsAvailable), ListActions,
        NextNodes)

successor(_, [], []).
successor(nodo(F, G, Node, ActionsAvailable), [Action | OtherActions],
    Successors) :-
    trasforma(Action, Node, NewNode),
    append(ActionsAvailable, [Action], NewActionAvailable),
    successor(nodo(F, G, Node, ActionsAvailable), OtherActions,
        OtherNodes),
    g(G, Node, NewNode, NewG),
    h(NewNode, NewH),
    NewF is NewG + NewH,
    ord_add_element(OtherNodes, nodo(NewF, NewG, NewNode,
        NewActionAvailable), Successors).

astar :-
```

```

iniziale(Node),
h(Node, H),
G is 0,
F is G + H,
astar_search([nodo(F, G, Node, [])], [], PathSolution),
write(PathSolution).

```

Il punto di entrata è la regola *astar*, che per essere vera deve soddisfare tutti i goal del suo corpo:

- deve esistere il nodo *iniziale(Node)*;
- deve esistere l'euristica *h(Node, H)* del nodo corrente;
- deve esistere *G* uguale a zero;
- deve esistere *F* uguale alla somma di *G* e *H*;
- *astar_search/3* deve essere true;
- *write(PathSolution)* stampa a video il cammino trovato;

La regola *astar_search/3* è utilizzata per la ricerca ricorsiva della soluzione. Ne esistono due, una per il caso base, e l'altra per la ricorsione.

La prima (indicata con %1 nel listato) è vera se il nodo attuale è quello finale. In tal caso la ricerca è finita e l'algoritmo termina scrivendo il cammino trovato. Da notare come in questo caso non sia presente la *CUT*, che invece era stata utilizzata nella ricerca *Iterative deepening*. La seconda, invece, (indicata con %2 nel listato) è vera:

- se il nodo attuale fa parte della lista dei nodi chiusi, allora la ricerca riparte escludendo l'attuale nodo preso in esame perché già chiuso; In caso contrario, la presenza del ";" permette di passare a valutare il secondo goal;
- se si espande a partire dal nodo attuale;
- se le liste dei nodi e dei vicini del nodo attuale vengono unite in nuova lista;
- riparte la ricerca con la nuova frontiera, inserendo il vecchio nodo nella lista dei nodi chiusi e con il cammino finora percorso;

La regola *expand/2* permette di espandere l'attuale nodo; nello specifico:

- il comando *findall/3* restituisce in una lista tutte le azioni possibili dal nodo attuale;
- *successor/3* cerca i nodi raggiungibili dall'attuale nodo con le azioni individuate precedentemente;

La regola *successor/3*, anche in questo caso, ha un caso base e una chiamata ricorsiva. La prima, in realtà, è semplicemente un fatto che sancisce che quel nodo non ha figli. La regola ricorsiva, invece, vale negli altri casi ed è vera se:

- l'azione individuata porta ad un nuovo nodo;
- si richiama la funzione *successor/3* dal nodo attuale con le restenanti azioni trovate sui nodi successivi;
- viene calcolata la *g* del nuovo nodo;
- viene calcolata la *h* dal nuovo nodo;
- viene calcolata la nuova *f* ;
- tutti i nodi individuati vengono aggiunti alla lista dei nodi successori;

2.2.1.1 Euristica della metropolitana di Londra

L'euristica utilizzata nella metropolitana di Londra è la distanza Euclidea. Tale scelta è dovuta al fatto che la distanza euclidea è un'euristica ammissibile, dato che essa rappresenta la distanza minima tra due punti.

2.2.2 Mondo dei blocchi

In questa sezione viene riportata l'implementazione della strategia di ricerca *A** per il dominio del mondo dei blocchi. Come nel caso dell'*Iterative deepening*, l'algoritmo è il medesimo utilizzato per la metropolitana di Londra. La soluzione è divisa però in quattro blocchi, data la presenza del modulo dell'euristica:

- *3_astar_block.pl* contiene regole e fatti che realizzano la strategia;
- *action_block.pl* contiene le azioni possibili;
- *heuristic_block.pl* contiene le euristiche utilizzate su questo dominio nella ricerche informate;

- *configuration_block.pl* contiene i fatti e le regole che descrivono il dominio (blocchi, stato finale e stato iniziale); anche in questo caso due configurazioni sono state fornite con diversa complessità;

A causa della complessità del problema è stato necessario modificare la grandezza massima dello stack di memoria a disposizione. La configurazione di default, infatti, non era sufficiente, facendo fallire la ricerca per *Out of local stack*.

Questo accadeva solo con la configurazione con un maggiore numero di blocchi, mentre l'altra restituiva la soluzione in modo corretto. L'istruzione riportata nel listato ci consente di aumentare il numero di *stacks* a disposizione, consentendo così la terminazione in modo corretto dell'esecuzione del programma.

```
: - set_prolog_stack(global, limit(12 000 000 000)).
```

2.2.2.1 Euristica del mondo dei blocchi

La scelta è ricaduta in una differenza insiemistica, ovvero il numero di stati differenti che ci sono in due liste ordinate; i stati considerati sono lo stato goal e lo stato attuale.

```
length_list([], 0).
length_list([_|List], Length) :-  
    length_list(List, X),  
    Length is X + 1.

heuristic1(List1, List2, Card) :-  
    ord_subtract(List1, List2, DifferenceList),  
    length_list(DifferenceList, Length),  
    Card is abs(Length) - 1.

h(N, Distance) :-  
    goal(G),  
    heuristic1(N, G, Distance).

g(G, NewG) :-  
    NewG is G + 1.
```

Per ottenerla, le due liste contenti le due situazioni (quella attuale e quella del goal) vengono sottratte in modo ordinato, ottenendo una terza lista con gli stati differenti che

sono presenti nella lista *goal* ma non nella lista attuale; la lunghezza di questa nuova lista viene calcolata e questa diventa la misura della distanza che separa la configurazione dei blocchi nello stato attuale rispetto a quello del *goal*. Riguardo invece la funzione g essa è semplicemente il numero di azioni effettuate per passare da uno stato all'altro (con stato si deve intendere, in questo caso, la lista di blocchi disposti sul tavolo), quindi si è deciso di utilizzare una funzione che incrementa ogni volta di una unità.

2.3 IDA*

*IDA** è un strategia di ricerca informata che unisce le caratteristiche di *Iterative deepening* e A^* . Dal primo prende la ricerca in profondità limitata incrementale e dal secondo la misura utilizzata per limitare la ricerca, cioè la funzione f . Quando non viene trovata una soluzione alla profondità fissata, allora questa viene incrementata; più nello specifico: nel momento in cui la ricerca fallisce alla profondità stabilità, viene ricalcolata f e scelto il valore minimo tra tutti gli *f-values* che superano il valore di soglia precedente.

La definizione $g(n)$ e $h(n)$ e $f(n)$ è la medesima di A^* , quindi valgono le stesse considerazioni. Come per A^* , *IDA** garantisce di trovare il cammino minimo se l'euristica utilizzata è ammmissibile e il *branching factor* finito, quindi è una strategia completa e ottima.

Asintoticamente, in termini di tempo, *IDA** si comporta allo stesso modo di A^* e *Iterative deepening*, mentre in termini di spazio *IDA** fa meglio di A^* perché non si tiene traccia dei nodi che si intende visitare, e ricorda solo i nodi del cammino corrente.

Di contro, *IDA** deve rigenerare i cammini, proprio perché solo un cammino alla volta è tenuto in memoria; rigenerare cammini può portare ad un notevole *overhead*, per questo motivo si tende a preferire A^* quando è spesso necessario espandere un nuovo percorso perché i valori di f sono molto ravvicinati tra di loro.

2.3.1 Metropolitana di Londra

In questa sezione viene riportata l'implementazione della strategia di ricerca *IDA** applicata al dominio della metropolitana di Londra. Come nel caso precedente, la soluzione è stata divisa in tre moduli differenti, due dei quali sono i medesimi di A^* . Infatti in entrambe le strategie viene utilizzata la stessa euristica.

Il file che contiene il modulo principale, cioè l'algoritmo, è `2_idastar_tube.pl` ed è riportato nel listato. Il funzionamento, in virtù di quanto detto poc'anzi, non si allontana molto

dall'implementazione di *Iterative deepening*, perché si tratta di una ricerca in profondità, ma con delle differenze:

- la ricerca in profondità controlla sempre se la f attuale è minore della profondità raggiunta; se lo è, allora puoi continuare la ricerca, altrimenti è necessario aggiornare la profondità al minimo valore di f che supera la soglia attuale;
- in *IDA** c'è da tenere in considerazione l'euristica, che nell'*Iterative deepening* non era presente, quindi f , g e h ; di fatto questa parte di aggiornamento è identica a quella di A^* ;
- l'aggiornamento della profondità viene realizzata dalla regola *update_cut_depth/1*, dove attraverso *CUT* e *backtracking* si assegna la minore f alla soglia;
- l'utilizzo del fatto $f_min(99999)$ e del comando `:dynamic(f_min/1)` consente, il primo, di inizializzare il *bound* ad un valore molto alto (è come se fosse infinito), così da poter essere aggiornato alla prima iterazione; il secondo, invece, avverte il motore inferenziale che il suo argomento potrà essere modificato durante l'esecuzione attraverso l'utilizzo dei comandi *asserta/1* e *retract/2*. Saranno proprio questi due comandi, nella regola *update_cut_depth/1* ad aggiornare la soglia;
- come in *Iterative deepening*, il caso base della ricorsione della ricerca in profondità, cioè la regola *dfs_bound/6*, presenta la *CUT* che si rende necessaria dato quello che si è detto sulla gestione dei cammini da parte di *IDA**;

```

:- use_module(action_tube).
:- use_module(heuristic_tube).

dfs_bound(Node, _, [], F, _, CutDepth) :-
    F =< CutDepth,
    finale(Node), !.

dfs_bound(Node, Visited, [Action | OtherAction], F, G, CutDepth) :-
    F =< CutDepth,
    applicabile(Action, Node),
    trasforma(Action, Node, NewNode),
    \+ member(NewNode, Visited),
    g(G, Node, NewNode, NewG)

```

```
h(NewNode, NewH),
NewF is NewG + NewH,
dfs_bounded(NewNode, [Node | Visited], OtherAction, NewF, NewG,
CutDepth).

dfs_bounded(_, _, _, F, _, CutDepth) :-
    F > CutDepth,
    update_cut_depth(F),
    fail.

update_cut_depth(F) :-
    f_min(Bound),
    Bound =< F, !
    ;
    retract(f_min(Bound)), !,
    asserta(f_min(F)).
:- dynamic(f_min/1).
f_min(99999).

idastar_search(StartNode, PathSolution, F, G, CutDepth) :-
    dfs_bounded(StartNode, [], PathSolution, F, G, CutDepth).

idastar_search(StartNode, PathSolution, F, G, _) :-
    f_min(NewCutDepth),
    retract(f_min(NewCutDepth)),
    asserta(f_min(99999)),
    idastar_search(StartNode, PathSolution, F, G, NewCutDepth).

idastar :-
    iniziale(Node),
    h(Node, H),
    G is 0,
    F is G + H,
    idastar_search(Node, PathSolution, F, G, F),
    write(PathSolution).
```

2.3.2 Mondo dei blocchi

In questa sezione viene presa in esame l'implementazione della strategia di ricerca *IDA** per il mondo dei blocchi.

Ancora una volta, la soluzione è la medesima riportata nella sezione dedicata alla metropolitana di Londra ed è contenuta nel file *2_idastar_block.pl*. Inoltre, tutte le considerazioni riguardanti l'euristica fatte per il mondo dei blocchi e *A** valgono anche per questa strategia di ricerca, quindi si eviterà di farle nuovamente.

Capitolo 3

Risultati

In questo terzo e ultimo capitolo sono riportati i risultati ottenuti con le strategie di ricerca illustrate nel secondo capitolo applicate ai domini descritti nel primo capitolo.

3.1 METROPOLITANA DI LONDRA

Qui riportiamo i risultati riguardanti il dominio della metropolitana di Londra.

3.1.1 Iterative deepening

La prima soluzione trovata da *Iterative deepening* è:

```
[sali(circle,0),vai(circle,0,Bayswater,Paddington),vai(circle,0,  
Paddington,Baker Street),vai(circle,0,Baker Street,Kings Cross),scendi  
(Kings Cross),sali(piccadilly,0),vai(piccadilly,0,Kings Cross,Holborn)  
,vai(piccadilly,0,Holborn,Covent Garden),scendi(Covent Garden)]
```

3.1.2 A*

La prima soluzione trovata da *A** è:

```
[sali(circle,0),vai(circle,0,Bayswater,Paddington),scendi(Paddington),  
sali(bakerloo,0),vai(bakerloo,0,Paddington,Baker Street),vai(bakerloo  
,0,Baker Street,Oxford Circus),vai(bakerloo,0,Oxford Circus,Piccadilly  
Circus),scendi(Piccadilly Circus),sali(piccadilly,1),vai(piccadilly  
,1,Piccadilly Circus,Leicester Square),vai(piccadilly,1,Leicester  
Square,Covent Garden),scendi(Covent Garden)]
```

3.1.3 IDA*

La prima soluzione trovata da *IDA** è:

```
[sali(circle,0),vai(circle,0,Bayswater,Paddington),scendi(Paddington),
sali(bakerloo,0),vai(bakerloo,0,Paddington,Baker Street),vai(bakerloo
,0,Baker Street,Oxford Circus),vai(bakerloo,0,Oxford Circus,Piccadilly
Circus),scendi(Piccadilly Circus),sali(piccadilly,1),vai(piccadilly
,1,Piccadilly Circus,Leicester Square),vai(piccadilly,1,Leicester
Square,Covent Garden),scendi(Covent Garden)]
```

3.2 MONDO DEI BLOCCHI

Qua riportiamo i risultati riguardanti il dominio del mondo dei blocchi.

3.2.1 Iterative deepening

La prima soluzione trovata da *Iterative deepening* è:

```
[pickup(d),stack(d,e),pickup(c),stack(c,d),unstack(a,b),putdown(a),pickup
(b),stack(b,c),pickup(a),stack(a,b)]
```

3.2.2 A*

La prima soluzione trovata da *A** è:

```
[pickup(d),stack(d,e),pickup(c),stack(c,d),unstack(a,b),stack(a,f),pickup
(b),stack(b,c),unstack(a,f),stack(a,b)]
```

3.2.3 IDA*

La prima soluzione trovata da *IDA** è:

```
[pickup(d),stack(d,e),pickup(c),stack(c,d),unstack(a,b),putdown(a),pickup
(b),stack(b,c),pickup(a),stack(a,b)]
```