



Progetto d'esame per la parte programmazione logica/ASP

Michele Colombino



Attività 1 - PROLOG

- ❖ Labirinto con più uscite (KB)
- ❖ Strategie di ricerca blind e informate
- ❖ Test e risultati finali

Base di Conoscenza

Nome File	Dominio		Azioni
Obiettivi	il file è utile per definire il concetto di labirinto		il file descrive le possibili azioni eseguibili
Codice	<pre>% Labirinto 5x5 con multiple uscite num_righe(5). num_colonne(5). iniziale(pos(1,1)). finale(pos(3,5)). finale(pos(4,2)). occupata(pos(1,3)). occupata(pos(2,3)). occupata(pos(3,1)). occupata(pos(4,3)). occupata(pos(4,4)). occupata(pos(5,3)).</pre>		<pre>applicabile(est, pos(Riga, Colonna)):- num_colonne(NC), Colonna < NC, NuovaColonna is Colonna + 1, \+occupata(pos(Riga, NuovaColonna)). applicabile(sud, pos(Riga, Colonna)):- num_righe(NR), Riga < NR, NuovaRiga is Riga + 1, \+occupata(pos(NuovaRiga, Colonna)). applicabile(ovest, pos(Riga, Colonna)):- Colonna > 1, NuovaColonna is Colonna - 1, \+occupata(pos(Riga, NuovaColonna)). applicabile(nord, pos(Riga, Colonna)):- Riga > 1, NuovaRiga is Riga - 1, \+occupata(pos(NuovaRiga, Colonna)). muovi(est, pos(Riga, Colonna), pos(Riga, NuovaColonna)):- NuovaColonna is Colonna + 1. muovi(sud, pos(Riga, Colonna), pos(NuovaRiga, Colonna)):- NuovaRiga is Riga + 1. muovi(ovest, pos(Riga, Colonna), pos(Riga, NuovaColonna)):- NuovaColonna is Colonna - 1. muovi(nord, pos(Riga, Colonna), pos(NuovaRiga, Colonna)):- NuovaRiga is Riga - 1.</pre>

Iterative deepening

- ❖ Si imposta la profondità massima di ricerca (in genere si inizializza a 0), quindi si avvia la ricerca.
- ❖ Se si raggiunge la condizione di terminazione, l'algoritmo restituisce il percorso e termina.
- ❖ Se non viene trovato il percorso di uscita entro la profondità massima, si incrementa la profondità massima di 1.
- ❖ Si ripete la ricerca fin quando non viene trovato il percorso di uscita.

```
prova(Solu):-
    id(Solu, 0).

id(Soluzione, Soglia):-
    depth_limit_search(Soluzione, Soglia),
    costoPassi(Soluzione, CostoCammino),
    write(Soluzione),
    write(CostoCammino).

id(Soluzione, Soglia):-
    NuovaSoglia is Soglia + 1,
    num_righe(NR),
    num_colonne(NC),
    SogliaMax is (NR * NC) / 2,
    NuovaSoglia < SogliaMax,
    id(Soluzione, NuovaSoglia).

depth_limit_search(Soluzione, Soglia):-
    iniziale(S),
    dfs_aux(S, Soluzione, [S], Soglia).

dfs_aux(S, [], _, _) :-
    finale(S).

dfs_aux(S, [Azione|AzioniTail], Visitati, Soglia):-
    Soglia>0,
    applicabile(Azione, S),
    muovi(Azione, S, SNuovo),
    \+member(SNuovo, Visitati),
    NuovaSoglia is Soglia-1,
    dfs_aux(SNuovo, AzioniTail, [SNuovo|Visitati], NuovaSoglia).
```

Ricerca informata con IDA*

- ❖ IDA* con euristica "**distanza di Manhattan**" (predicato "uscitaEuristica"). La ricerca parte dalla regola "**prova**"
- ❖ Le azioni hanno un costo, un predicato **aggiorna_costo_min** ricorda il costo minimo trovato durante la ricerca.
- ❖ La regola **limiteIda** gestisce i casi in cui è necessario cambiare la soglia di costo per la ricerca.
- ❖ Usando un approccio ricorsivo si confrontano le distanze correnti con quelle trovate in precedenza. Infine si calcola il costo totale del percorso e il percorso migliore viene restituito come risultato

```
prova(Sol):-
    iniziale(Start),
    uscitaEuristica(Start, SogliaIniziale),
    num_righe(NR),
    num_colonne(NC),
    FCMax is NR * NC,
    assert(costo_min(FCMax)),
    limiteIda(Soluzione, SogliaIniziale, Start),
    reverse(Sol, Soluzione),
    costoTot(Soluzione, CostoCammino),
    write(Soluzione),
    write(CostoCammino).

limiteIda(Soluzione, Soglia, S_Attuale):-
    ida_C(node(S_Attuale, [], Soglia), Soluzione, [S_Attuale], Soglia).

limiteIda(Soluzione, Soglia, S_Attuale):-
    costo_min(NuovaSoglia),
    retract(costo_min(NuovaSoglia)),
    num_righe(NR),
    num_colonne(NC),
    FCMax is NR * NC,
    assert(costo_min(FCMax)),
    \+NuovaSoglia = Soglia,
    limiteIda(Soluzione, NuovaSoglia, S_Attuale).

ida_C(node(S_Attuale, Azioni, _), Azioni, _, _):-
    finale(S_Attuale).
```

Ricerca informata con A*

- ❖ A* con euristica **distanza di Euclide**. La ricerca parte dalla regola **prova**. una seconda regola **confrontoeuristico** calcola l'euristica della posizione iniziale.

- ❖ La regola **aStar** riceve una lista contenente un nodo con la posizione iniziale, l'euristica calcolata e una lista vuota di azioni eseguite, da qui vengono **gestiti 3 casi**:

1. La posizione attuale coincide con quella finale
2. Si ricavano le azioni applicabili e si generano i nuovi nodi figli usando il nodo attuale i nodi visitati e la coda attuale dei nodi. infine si invoca ricorsivamente **aStar** con la nuova coda dei nodi.
3. La ricerca termina senza trovare una soluzione.

- ❖ In caso non ci siano più azioni applicabili si ritorna la coda finale per la prossima ricerca. Se l'azione genera uno stato già visitato, in questo caso l'azione viene ignorata.

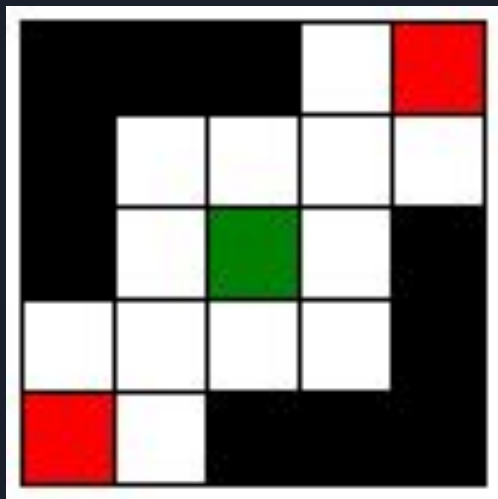
```
prova(Sol):-
    iniziale(Start),
    confrontoeuristico(Start, Costo_Euristica),
    aStar([node(Start, Costo_Euristica, [])], [], Sol),
    reverse(SolOrd, Sol),
    costoPassi(SolOrd, CostoNpassi),
    write(SolOrd),
    write(CostoNpassi).

Star([node(Start, _, AzioniEseguite)|_], _, AzioniEseguite):-finale(Start).

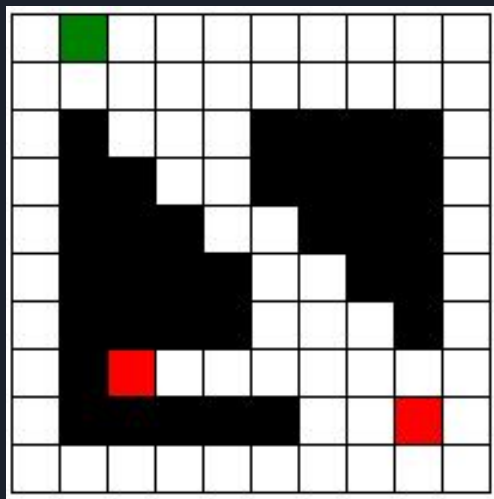
Star([node(Start, CostoFinale, AzioniEseguite)|Coda], Visitati, SolParziale):-
    findall(Azione, applicabile(Azione,Start), Applicabili),
    nuoviFigli(node(Start, CostoFinale, AzioniEseguite), Applicabili, [Start|Visitati],
    aStar(NuovaCoda, [Start|Visitati], SolParziale).

Star([], _, _):-
    write("Nessun percorso trovato").
```

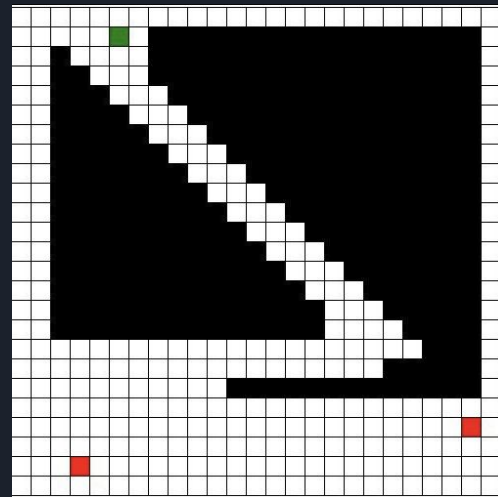
Test e risultati finali



Labirinto 5 x 5



Labirinto 10 x 10



Labirinto 25 x 25

Test e risultati finali

Test eseguiti su una
macchina con processore
i7-11370H a 3.30GHz


Per valutare le prestazioni degli algoritmi abbiamo usato le metriche presenti nel modulo “**statistics**” (predicato “**time**”). Due indicatori:

- **Tempo di esecuzione**
- **Numero di inferenze** eseguite

I risultati in tabella si riferiscono al tempo richiesto per la generazione della **prima soluzione**, avendo più **vie di uscita**.

Come si può notare da una prima analisi, l'algoritmo **A*** all'aumentare della complessità dello spazio di ricerca, restituisce le migliori performance rispetto alle altre strategie

	Iterative deepening	IDA*	A*
Labirinto 5 x 5	1,114 inferences, 0.000 CPU 0.000 sec	912 inferences, 0.000 CPU 0.000 sec	1,382 inferences, 0.000 CPU 0.000 sec
Labirinto 10 x 10	413,862 inferences, 0.031 CPU 0.036 sec	70,125 inferences, 0.000 CPU 0.011 sec	29,998 inferences, 0.000 CPU 0.006 sec
Labirinto 25 x 25	845,826,625 inferences, 163.750 CPU 165.773 sec	96,292 inferences, 0.016 CPU 0.013 sec	49,788 inferences, 0.016 CPU 0.010 sec



Attività 2 - CLINGO

- ❖ Calendario di un campionato di calcio
- ❖ Vincoli obbligatori e facoltativi
- ❖ Test e risultati finali

Vincoli obbligatori


- 1) Sono iscritte 20 squadre (**squadra**)
- 2) Il campionato prevede 38 giornate, 19 di andata e 19 di ritorno (**turno_andata**, **turno_ritorno**) Inoltre, le giornate di andata e di ritorno **NON sono simmetriche**
- 3) Ogni squadra fa riferimento ad una città, (**in_città**) che offre la struttura in cui la squadra gioca gli incontri in casa. (***milan e inter** condividono lo stesso stadio*).

```
% 1) Squadre
squadra(milan; inter; lazio; juventus ...

% 2) Giornate: andata, ritorno
turno(1..38).
turno_andata(1..19).
turno_ritorno(20..38).

% 3) Ogni squadra fa riferimento ad una città
in_città( milan, stadio_inter_milan;
          inter, stadio_inter_milan;
          lazio, stadio_lazio;
          ...

:- partita(andata,SquadraA,SquadraB, _, T1),
   partita(ritorno,SquadraB,SquadraA, _, T2),
   T2 == T1 + 19.
```

- 
4. Ogni squadra affronta due volte tutte le squadre, una volta in casa e una volta fuori casa

- ❖ Il predicato **“partita”** modella un singolo incontro sportivo. Ogni squadra si sfida all'andata e al ritorno un numero di volte pari al numero di squadre totali / 2
- ❖ Nella stessa giornata, una squadra non può giocare 2 volte
- ❖ Non può capitare che due squadre giocano andata e ritorno solo all'andata o solo al ritorno

```
10 {partita(andata, SquadraA, SquadraB, Citta, N):  
    squadra(SquadraA),  
    squadra(SquadraB),  
    in_citta(SquadraA,Citta),  
    SquadraA <> SquadraB} 10 :- turno_andata(N).  
  
:- squadra(Squadra),  
   turno(Turno),  
   Num_A = #count{A: partita(_,Squadra,A,_, Turno)},  
   Num_B = #count{B: partita(_,B,Squadra,_, Turno)},  
   Num_A + Num_B <> 1.  
  
:- partita(Tipo, SquadraA, SquadraB, _, _),  
   partita(Tipo, SquadraB, SquadraA, _, _).
```



Altri vincoli

- 5) Due delle 20 squadre sono nella stessa città e condividono lo stesso stadio, quindi non possono giocare entrambe in casa nella stessa giornata.

❖ Facoltativo 1

Ciascuna squadra non deve giocare mai più di due partite consecutive in casa o fuori casa

❖ Facoltativo 2

La distanza tra una coppia di gare di andata e ritorno è di almeno 10 giornate

```
:- partita(Tipo, Squadra1, _, C1, N),  
   partita(Tipo, Squadra2, _, C2, N),  
   Squadra1 <> Squadra2,  
   C1 == C2.
```

```
:- partita(_, Squadra, _, _, T1),  
   partita(_, Squadra, _, _, T1+1),  
   partita(_, Squadra, _, _, T1+2).
```

```
:- partita(_, _, Squadra, _, T1),  
   partita(_, _, Squadra, _, T1+1),  
   partita(_, _, Squadra, _, T1+2).
```

```
:- partita(andata, SquadraA, SquadraB, _, T1),  
   partita(ritorno, SquadraB, SquadraA, _, T2),  
   T2 - T1 <= 10.
```

Test e risultati finali

Test eseguiti su una
macchina con processore
i7-11370H a 3.30GHz

- ❖ I test con i vincoli facoltativi hanno richiesto molto tempo per la determinazione del primo answer set.
- ❖ All'aumentare del numero delle squadre il tempo di esecuzione cresce.
- ❖ Gli ultimi test non hanno restituito una risposta in tempi ragionevoli.

Test con vincoli obbligatori						
	6 Squadre (10 turni)	10 Squadre (18 turni)	12 Squadre (22 turni)	14 Squadre (26 turni)	18 Squadre (34 turni)	20 Squadre (38 turni)
Time	0.026s	1.625s	34.866s	85.214s	1288.533s	?
CPU TIME	0.031s	1.344s	28.125s	39.453s	907.563s	?

Test con vincoli facoltativi						
	6 Squadre (10 turni)	10 Squadre (18 turni)	12 Squadre (22 turni)	14 Squadre (26 turni)	18 Squadre (34 turni)	20 Squadre (38 turni)
Time	0.033s	3.636s	98.478s	153.554s	?	?
CPU TIME	0.016s	3.109s	81.766s	147.313s	?	?



Grazie per l'attenzione