Prolog

Progetto di Intelligenze Artificiali e Laboratorio - Parte 1

Emanuele Gentiletti, Alessandro Caputo

Indice

Introduzione

La consegna del progetto prevedeva l'implementazione degli algoritmi Iterative Deepening, A* e IDA*, e di provarli in diversi contesti in uno stesso dominio. In questa relazione trattiamo il codice scritto per gli algoritmi e i risultati che abbiamo ottenuto nella loro esecuzione in tre domini, con due diverse euristiche. Il dominio che abbiamo scelto nella trattazione è il mondo dei blocchi.

Ricerca non informata

Iterative Deepening

Per l'implementazione dell'algoritmo di Iterative Deepening, abbiamo riutilizzato il codice riguardante la ricerca in profondità visto a lezione, in particolare il predicato `depth_limit_search`.

```
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),
    trasforma(Azione, S, SNuovo),
    \+ member(SNuovo, Visitati),
    NuovaSoglia is Soglia-1,
    dfs_aux(SNuovo, AzioniTail, [SNuovo|Visitati], NuovaSoglia).
```

`depth_limit_search` esegue una ricerca in profondità limitata a partire dallo stato iniziale. Nell'implementazione dell'Iterative Deepening lo andiamo a richiamare aumentando la soglia di un'unità per ogni iterazione. Permettiamo all'utente di stabilire una soglia massima, oltre la quale l'algoritmo termina se non ha ancora trovato una soluzione.

```
iterative_deepening(Soluzione, SogliaMax) :-
    iterative_deepening_aux(Soluzione, 1, SogliaMax).

iterative_deepening_aux(Soluzione, SogliaCorrente, _) :-
    depth_limit_search(Soluzione, SogliaCorrente).

iterative_deepening_aux(Soluzione, SogliaCorrente, SogliaMassima) :-
    SogliaSuccessiva is SogliaCorrente+1,
    SogliaSuccessiva=<SogliaMassima,
    iterative_deepening_aux(Soluzione, SogliaSuccessiva, SogliaMassima).
```

`iterative_deepening` funge da interfaccia utente per il predicato. Accetta come parametri la soluzione e la soglia massima. Il predicato ausiliario `iterative_deepening_aux` accetta come ulteriore parametro `SogliaCorrente`, che rappresenta la profondità di esplorazione nell'iterazione corrente.

L'algoritmo inizia chiamando `iterative_deepening_aux` con soglia corrente 1.
`iterative_deepening_aux` esegue la ricerca in profondità con soglia `SogliaCorrente`. Se non trova soluzione, scatta la clausola successiva, che esegue i seguenti passi:

- unifica `SogliaSuccessiva` con il numero successivo di `SogliaCorrente`
- controlla se `SogliaSuccessiva <= SogliaMassima`. Se sì, richiama ricorsivamente `iterative_deepening_aux` con la soglia incrementata.

In questo modo, l'algoritmo procede fino a trovare una soluzione, o fino a terminare per aver superato la soglia definita dall'utente.

Ricerca informata

Nell'implementazione degli algoritmi di ricerca informata, abbiamo anche tenuto in conto del fatto che i costi delle azioni potrebbero differire tra loro, e non essere necessariamente unitari. Di fatto, per queste scelte implementative, l'algoritmo di Iterative Deepening trova la soluzione ottimale in base al numero di azioni, mentre A* e IDA* trovano la soluzione ottimale in base al costo delle azioni (a patto che l'euristica usata sia ammissibile e consistente).

A*

Nell'implementazione di A*, inizialmente procediamo come nella ricerca in ampiezza. Per rappresentare gli stati usiamo dei termini composti `nodo(F, C, Stato, Azioni)`, dove i parametri stanno a indicare:

- `Stato`: La struttura corrente dello stato.
- `Azioni`: La lista di azioni eseguite per arrivare allo stato corrente.
- `C`: La somma dei costi di `Azioni`.
- `F`: Il valore di `C` più l'euristica a partire da `Stato`.

Salviamo i valori `C` ed `F` per poter implementare il criterio di ricerca dell'algoritmo, ovvero $f(x) = g(x) + h(x)$, dove il valore di $f(x)$ corrisponde a quello di `F`. Per poter ordinare i nodi in base a `F`, lo poniamo come primo parametro del funtore.

Il corpo dell'algoritmo inizia da `a_star_aux`. Questa regola utilizza 3 parametri:

1. Una lista di nodi da visitare.
2. Una lista di nodi visitati.
3. La soluzione.

```
% a_star_aux(DaVisitare, Visitati, Soluzione)
a_star_aux(
  [nodo(CostoPiuEuristica, Costo, S, Azioni)|Tail],
  Visitati,
  Soluzione
) :-
    ...
```

Per prima cosa cerchiamo tutte le azioni applicabili per il primo stato nella coda attraverso il predicato `findall` e le inseriamo nella lista degli applicabili.

In seguito chiamiamo la regola genera_figli che genera i figli del nodo corrente in base agli applicabili trovati. Questi nodi vengono poi inseriti nella coda dei nodi da visitare.

```
  findall(Applicabile, applicabile(Applicabile, S), ListaApplicabili),
  genera_figli(
    nodo(CostoPiuEuristica, Costo, S, Azioni),
    ListaApplicabili,
    [S|Visitati],
    ListaFigli
  ),
  append(Tail, ListaFigli, NuovaCoda),
```

A questo punto i nodi vengono riordinati tramite list_to_ord_set, in modo che si trovino in ordine crescente rispetto a $f(x)$. Infine richiamiamo ricorsivamente a_star_aux con la nuova coda dei nodi da visitare.

```
  list_to_ord_set(NuovaCoda, NuovaCodaOrdinata),
  a_star_aux(NuovaCodaOrdinata, [S|Visitati], Soluzione).
```

Analizziamo ora la regola genera_figli. Questa regola contiene 4 parametri:

1. Il nodo di cui calcolare i figli.
2. La lista degli applicabili per lo stato corrente .
3. La lista degli stati già visitati.
4. La lista di output dei figli generati.

```
genera_figli(
  Nodo,
  [Applicabile|AltriApplicabili],
  Visitati,
  [Figlio|FigliTail]
) :-
  Nodo=nodo(_, Costo, S, AzioniPerS),
  ...
```

Abbiamo scelto di far unificare gli argomenti di nodo nel corpo della regola, al fine di renderla più leggibile.

La regola applica l’azione in testa alla lista degli applicabili, generando il nodo figlio con i suoi valori di $g(x)$, $h(x)$ e $f(x)$.

```
  trasforma(Applicabile, S, SNuovo),
  \+ member(SNuovo, Visitati),
  !,
  euristica(SNuovo, Euristica),
  costo(Applicabile, CostoApplicabile),
  CostoFiglio is CostoApplicabile+Costo,
  CostoPiuEuristicaFiglio is CostoFiglio+Euristica,
  Figlio=nodo(
    CostoPiuEuristicaFiglio,
    CostoFiglio,
    SNuovo,
    [Applicabile|AzioniPerS]
  ),
```

La regola viene poi richiamata ricorsivamente con le azioni applicabili rimanenti.

```
  genera_figli(Nodo, AltriApplicabili, Visitati, FigliTail).
```

Se l'azione applicabile porta a uno stato già visitato la regola fallirà a causa del controllo `\+ member(SNuovo, Visitati)`. In questo caso verrà verificata una seconda clausola di `genera_figli` che procede direttamente all'elaborazione del figlio successivo. Subito dopo il controllo abbiamo inserito un cut in modo tale da non permettere erroneamente il backtraking in questa clausola.

```
genera_figli(Nodo, [_|AltriApplicabili], Visitati, FigliTail) :-
    genera_figli(Nodo, AltriApplicabili, Visitati, FigliTail).
```

IDA*

L'implementazione di IDA* è simile a quella dell'Iterative Deepening. Ci sono due predicati principali: `ida_star` e `cost_limit_search` (e i loro corrispondenti predicati ausiliari). `cost_limit_search` ha il compito di eseguire la ricerca in profondità, mentre `ida_star` stabilisce la soglia massima di ogni chiamata di `cost_limit_search`.

Rispetto all'Iterative Deepening, c'è una differenza importante nell'implementazione della ricerca limitata. Se `cost_limit_search` non trova una soluzione una volta raggiunta la profondità massima, deve salvare le informazioni necessarie a stabilire la profondità di ricerca successiva. Abbiamo implementato questa feature usando il cut, un nuovo predicato `assert_prossima_soglia` e una nuova clausola su `dfs_aux`.

```
cost_limit_search(Soluzione, CostoMaxCammino) :-
    iniziale(S),
    dfs_aux(S, Soluzione, [S], 0, CostoMaxCammino).

dfs_aux(S, _, _, CostoCammino, CostoMaxCammino) :-
    CostoCammino>CostoMaxCammino,
    !,
    euristica(S, Euristica),
    Soglia is CostoCammino+Euristica,
    assert_prossima_soglia(Soglia),
    false.

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

In `dfs_aux`, invece di ricevere una soglia in input, si considerano il costo del cammino attuale e il costo massimo del cammino. Quando `dfs_aux` viene valutato con un `CostoCammino > CostoMaxCammino`, questo calcola la stima del costo per arrivare allo stato finale a partire dallo stato attuale, sommando `CostoCammino` al risultato della funzione euristica, per poi asserire il valore ottenuto nel predicato dinamico `prossima_soglia(S)`. Il predicato poi fallisce per permettere di continuare la ricerca.

Se invece il costo è ancora sotto il massimo, l'algoritmo procede normalmente con la ricerca limitata in profondità.

```
dfs_aux(
  S, [Azione|AzioniTail], Visitati, CostoCammino, CostoMaxCammino
) :-
  applicabile(Azione, S),
  trasforma(Azione, S, SNuovo),
  \+ member(SNuovo, Visitati),
  costo(Azione, CostoAzione),
  NuovoCostoCammino is CostoCammino+CostoAzione,
  dfs_aux(
    SNuovo, AzioniTail, [SNuovo|Visitati],
    NuovoCostoCammino, CostoMaxCammino
  ).
```

L'asserzione di `prossima_soglia` avviene tramite il predicato `assert_prossima_soglia(NuovaSoglia)`. Questo controlla se il valore di `NuovaSoglia` sia maggiore o uguale a un eventuale valore della soglia asserito in precedenza. Se sì, il predicato esegue un cut e non effettua altre operazioni, dato che ci interessa salvare solo il valore minimo tra quelli delle soglie trovate.

Se invece non c'è una soglia salvata in precedenza, o se la `NuovaSoglia` è minore di questa, `assert_prossima_soglia` procede a effettuare la `retract` della soglia precedente e ad asserire quella nuova.

```
assert_prossima_soglia(NuovaSoglia) :-
    prossima_soglia(SogliaPrecedente),
    NuovaSoglia>=SogliaPrecedente,
    !.

assert_prossima_soglia(NuovaSoglia) :-
    retractall(prossima_soglia(_)),
    assert(prossima_soglia(NuovaSoglia)).
```

Se alla fine della ricerca in profondità non è stata trovata una soluzione, l'algoritmo fa partire una nuova ricerca, usando come nuova soglia massima la soglia asserita in `prossima_soglia`. Questa logica è gestita all'interno di `ida_star_aux`:

```
ida_star_aux(Soluzione) :-
    prossima_soglia(Soglia),
    retract(prossima_soglia(_)),
    cost_limit_search(Soluzione, Soglia).

ida_star_aux(Soluzione) :- ida_star_aux(Soluzione).
```

In `ida_star_aux`, si trova la soglia tramite `prossima_soglia(Soglia)`, si ritrae il predicato `prossima_soglia` e si procede a effettuare la ricerca con limite di costo `Soglia`. Se `cost_limit_search` non trova una soluzione, `ida_star_aux` si richiama ricorsivamente, in modo che venga eseguita una nuova ricerca limitata nel costo con il nuovo valore di soglia trovato.

L'algoritmo parte con il predicato `ida_star(Soluzione)`, che asserisce la soglia al valore dell'euristica sullo stato iniziale e valuta il predicato `ida_star_aux(Soluzione)`.

```
ida_star(Soluzione) :-
    iniziale(S),
    euristica(S, SogliaIniziale),
    assert_prossima_soglia(SogliaIniziale),
    ida_star_aux(Soluzione).
```

Funzioni euristiche

Abbiamo implementato e confrontato due euristiche differenti. Nella formulazione delle euristiche siamo partiti da un concetto di fondo, ovvero il confronto tra gli insiemi che rappresentano gli stati. Dopo aver effettuato delle ricerche abbiamo deciso di implementare le seguenti euristiche:

Euristica 1: Calcola il numero di blocchi correnti che non sono nella posizione corretta

Euristica 2: Calcola la differenza tra stato corrente e stato finale considerando la posizione di ogni blocco rispetto al blocco sottostante e sovrastante. Se il blocco `A` nello stato finale dovrebbe trovarsi sopra il blocco `B` e sotto il blocco `C` e nello stato corrente si trova sotto il blocco `B` e sopra il blocco `C` allora aggiungiamo 2 al valore all'euristica.

Implementazione

L'idea è quella di considerare il risultato della sottrazione tra gli insiemi che descrivono lo stato finale e lo stato iniziale, in modo da ottenere tutti i fatti che differiscono tra i due stati. Considerando la cardinalità dell'insieme risultante, possiamo fare delle considerazioni sul numero di operazioni necessarie ad arrivare alla soluzione.

Nell'implementazione abbiamo considerato solo i fatti `ontable(X,Y)` e `on(X)`.

Euristica 1

```
euristica(StatoAttuale, Valore) :-
    goal(StatoFinale),
    ord_subtract(StatoFinale, StatoAttuale, DifferenzaStati),
    include(is_on, DifferenzaStati, StatiOn),
    length(StatiOn, Valore),
```

La regola ha due parametri, la soglia attuale e una variabile in cui inseriamo il valore dell'euristica calcolato. Per prima cosa effettuiamo la sottrazione tra insiemi attraverso la funzione `ord_subtract`, per poi selezionare i fatti che ci interessano (`on` e `ontable`) utilizzando la funzione `include`.

Abbiamo inserito inoltre due fatti che ci hanno permesso di inserire un unico parametro nella funzione `include` al fine di selezionare sia i fatti `on` che i fatti `ontable`.

```
is_on(on(_,_)).
is_on(ontable(_)).
```

Euristica 2

```
euristica(StatoAttuale, Valore) :-
    goal(StatoFinale),
    ord_subtract(StatoFinale, StatoAttuale, DifferenzaStati),
    include(is_on, DifferenzaStati, StatiOn),
    include(is_ontable, DifferenzaStati, StatiOntable),
    length(StatiOn, LunghezzaStatiOn),
    length(StatiOntable, LunghezzaStatiOntable),
    ValoreOn is LunghezzaStatiOn * 2,
    ValoreTable is LunghezzaStatiOntable,
    Valore is ValoreOn + ValoreTable.
```

L'implementazione della seconda euristica è molto simile a quella della prima. Di fatto ci è bastato incrementare di due il valore dell'euristica per ogni fatto `on` presente nell'insieme risultante dalla sottrazione dei due insiemi.

La strategia è quella di contare tutti i fatti `on` riguardanti un determinato cubo. Se nell'insieme risultante da `ordsubtract` ci fosse `on(A,B)on(C,B)` l'euristica incrementerebbe di due a causa del cubo `B`, perché si troverebbe nella posizione errata sia rispetto al cubo sovrastante sia rispetto al cubo sottostante. Successivamente incrementerebbe di uno per il cubo `A` e ancora di uno per il cubo `C`. Dal momento che ogni fatto `on` si riferisce a due cubi distinti ci basta incrementare l'euristica di due per ognuno dei fatti `on` presenti nell'insieme risultante dalla sottrazione.

Analisi

Abbiamo confrontato i tre algoritmi su tre domini e utilizzando due euristiche differenti. I parametri utilizzati per il confronto sono quattro:

- inferences: Il numero di inferenze effettuato dall'algoritmo in un esecuzione
- Execution Time: Tempo di un esecuzione in secondi
- Number of lips: Logical Inferences Per Second
- First Solution length: Il numero di passi presenti nella prima soluzione trovata.

Infine abbiamo confrontato i tre algoritmi su uno stesso dominio ma assegnando alle azioni un costo variabile.

Caratteristiche macchina utilizzata:

- SO Ubuntu 18.04 LTS
- Processore Intel Core i5 5200U
- Frequenza Massima 2,70 GHz
- Chache 3 MB
- RAM 8 GB

Dominio 1

Listing 1: Dominio 1: esempio moodle

```
stato iniziale    stato finale

                   a
 a                 b
 b d               c
 c e               d e
---------------    ------------
```

Tabella 1: Dominio 1, risultati con prima euristica

Algorithms	Inferences	Execution Time (s)	Lips	First Solution length
Iterative Deepening	3,331,110	0.367	9077721	12
A*	2,045,469	0,621	3292351	12
IDA*	2,862,494	0.325	8797702	12

Tabella 2: Dominio 1, risultati con seconda euristica

Algorithms	Inferences	Execution Time (s)	Lips	First Solution length
Iterative Deepening	3,331,110	0.367	9077721	12
A*	174,894	0.037	4682229	12
IDA*	5,930,528	0.651	9112884	12

Dominio 2

Listing 2: Dominio 2: esempio Prof. Torasso

```
stato iniziale      stato finale

                    a
                    b
                    c
a                   d
b c d e f g h       e
-------------       ------------
```

Tabella 3: Dominio 2, risultati con prima euristica

Algorithms	Inferences	Execution Time (s)	Lips	First Solution length
Iterative Deepening	1,536,727,053	187.355	8202216	10
A*	N/A	N/A	N/A	N/A
IDA*	10,766,835,266	1195.656	9004961	10

Tabella 4: Dominio 2, risultati con seconda euristica

Algorithms	Inferences	Execution Time (s)	Lips	First Solution length
Iterative Deepening	1,536,727,053	187.355	8202216	10
A*	37,874,888	23.033	1644371	10
IDA*	10,947,466,831	1203.364	9097384	10

Dominio 3

Listing 3: Dominio 3

```
stato iniziale      stato finale

a   d               b   e
b   e               c   f
c   f               a   d
--------------      ------------
```

Tabella 5: Dominio 3, risultati con prima euristica

Algorithms	Inferences	Execution Time (s)	Lips	First Solution length
Iterative Deepening	471,965,663	54.858	8603383	16
A*	669,712,233	563.830	1187792	16
IDA*	1,579,600,374	138.929	11369810	16

Tabella 6: Dominio 3, risultati con seconda euristica

Algorithms	Inferences	Execution Time (s)	Lips	First Solution length
Iterative Deepening	471,965,663	54.858	8603383	16
A*	5,587,396	2.168	2577590	16
IDA*	74,085,129	8.010	9249367	16

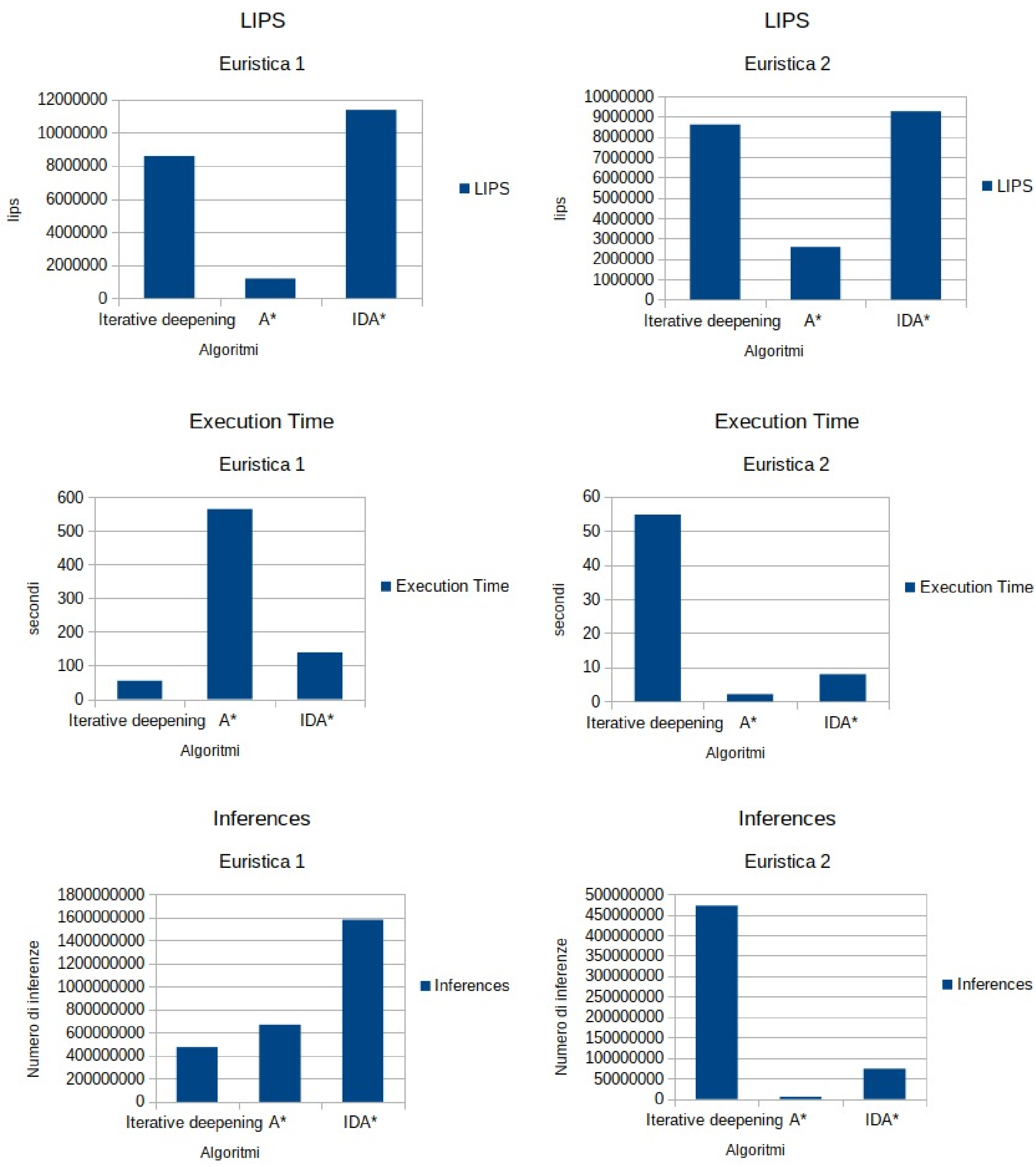

Figura 1: Confronto dei risultati per il terzo dominio

Confronto euristiche con costi variabili

Abbiamo assegnato costo 3 alle azioni `stack` e `unstack` e costo 1 alle azioni `putdown` e `pickup`.

Tabella 7: Dominio 3, risultati con prima euristica

Algorithms	Inferences	Execution Time (s)	Lips	First Solution length
A*	658,472,296	538.194	2577590	16
IDA*	593,054,438	69.742	8509952	16

Tabella 8: Dominio 3, risultati con seconda euristica

Algorithms	Inferences	Execution Time (s)	Lips	First Solution length
A*	283,752,336	31.995	8868703	16
IDA*	183,414,987	136.742	1341699	16

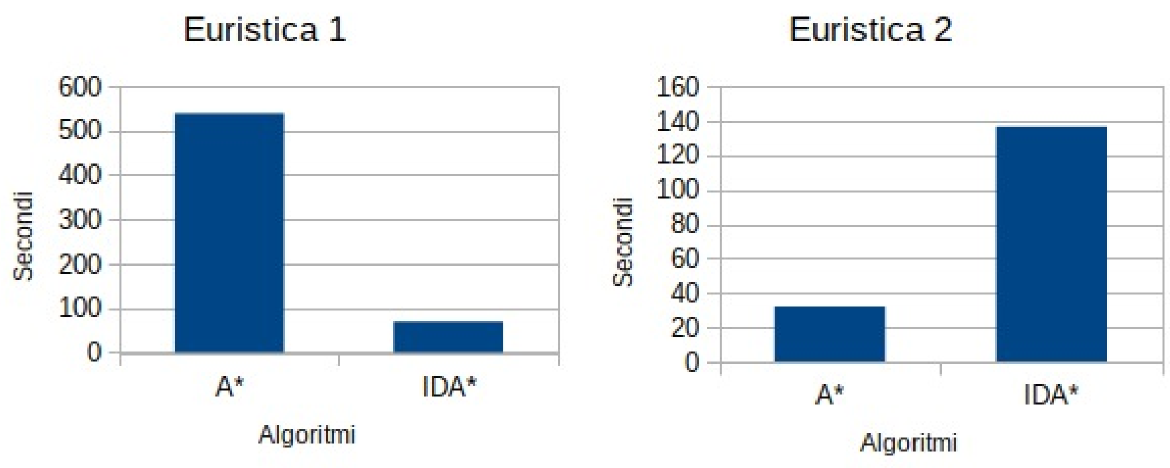

Figura 2: Confronto dei risultati per costo variabile

Osservazioni sui risultati

Gli istogrammi riportano le prestazioni degli algoritmi utilizzando le due euristiche. Il dominio di prova selezionato per la creazione degli istogrammi è il terzo perchè ha una complessità che si pone tra il primo e il secondo dominio.

Osservando le prestazioni degli algoritmi che effettuano una ricerca informata, notiamo come la seconda euristica approssimi molto meglio il numero di passi necessario per trovare la soluzione. Vediamo infatti una diminuzione significativa del numero di inferenze effettuate e quindi una diminuzione del tempo di esecuzione totale:

	Execution Time (Euristica 1)	Execution Time (Euristica 2)
A*	$\approx$ 9 minuti	$\approx$ 1.8 secondi
IDA*	$\approx$ 2,3 minuti	$\approx$ 7.8 secondi

Il tempo d'esecuzione dell'Iterative Deepening nel dominio tre è di $\approx$ 1 minuto, quindi si comporta meglio degli algoritmi a ricerca informata nel caso della prima euristica ma peggio nel caso della seconda euristica.

Questo dimostra come la prima euristica sia decisamente poco efficace. Di fatto rende i due algoritmi basati su ricerca informata meno efficenti di un algoritmo di ricerca non informata.

Per quanto riguarda il confronto degli algoritmi a ricerca informata per costi differenti, notiamo come questi causino un leggero miglioramento con l'utilizzo della prima euristica, ma un importante peggioramento con l'utilizzo della seconda euristica.

Aumentando il costo di determinate azioni, gli algoritmi a ricerca informata funzionano leggermente meglio utilizzando la prima euristica, mentre peggiorano decisamente utilizzando la seconda.

Un'ultima osservazione va fatta riguardo le dimensioni e le complessità dei domini testati. Le osservazioni fatte sugli algoritmi per il dominio 3 sono accettabili per tutti i domini ad eccezione del dominio due, ovvero il più complesso.

In questo caso, per quanto riguarda la prima euristica, non siamo riusciti ad ottenere dati riguardo l'algoritmo A*. l'esecuzione infatti è terminata a causa dell'esaurimento della memoria.

Per quanto riguarda l'utilizzo della seconda euristica invece, vediamo come IDA* abbia prestazioni stranamente peggiori tra tutti gli algoritmi, con un tempo di esecuzione di ben $\approx$ 20 minuti.