

GIOCO DELL'OTTO

*Tesina sulla ricerca - Intelligenza Artificiale
Cosimo Cecchi, Tommaso Visconti*

Indice

Cap. 1 - Introduzione.....	<i>pag. 3</i>
Cap. 2 - Il Gioco dell'8.....	<i>pag. 3</i>
Cap. 3 - Struttura del programma.....	<i>pag. 4</i>
Cap. 3.1 - Generazione degli stati iniziali.....	<i>pag. 4</i>
Cap. 3.2 - Calcolo dell'orbita.....	<i>pag. 4</i>
Cap. 3.3 - Algoritmi.....	<i>pag. 5</i>
Cap. 3.3.1 - Breadth-First Search.....	<i>pag. 5</i>
Cap. 3.3.2 - Depth-First Search.....	<i>pag. 6</i>
Cap. 3.3.3 - A*.....	<i>pag. 6</i>
Cap. 3.3.4 - IdA*.....	<i>pag. 7</i>
Cap. 3.4 - Euristiche.....	<i>pag. 7</i>
Cap. 3.4.1 - Tessere Fuori Posto.....	<i>pag. 8</i>
Cap. 3.4.2 - Manhattan.....	<i>pag. 8</i>
Cap. 4 - Analisi delle prestazioni.....	<i>pag. 8</i>
Cap. 4.1 - Penetranza.....	<i>pag. 8</i>
Cap. 4.2 - Fattore di ramificazione.....	<i>pag. 8</i>
Cap. 5 - Esecuzione del programma.....	<i>pag. 9</i>
Cap. 6 - Conclusioni.....	<i>pag. 10</i>

1 Introduzione

Il *problema della ricerca* rappresenta un'importante branca dell'Intelligenza Artificiale e si occupa dello studio di algoritmi in grado di trovare la soluzione ad un determinato problema nel modo più efficiente possibile, sempre che tale soluzione esista. I problemi che si affrontano tramite i metodi di ricerca sono tipicamente riconducibili al giungere ad uno stato prefissato, chiamato stato *goal*, a partire da una configurazione iniziale nota, muovendosi attraverso le possibili configurazioni che fanno convergere il problema alla soluzione; queste sono solitamente rappresentate all'interno di strutture ad albero, e la soluzione da cercare è proprio il cammino all'interno dell'albero, che conduce dalla radice al nodo che rappresenta la configurazione *goal*.

La difficoltà da affrontare nello studio dei possibili algoritmi risolutivi è rappresentata principalmente dall'ampiezza degli spazi di ricerca a confronto con una limitata capacità computazionale, il che rende indispensabile, se vogliamo giungere alla soluzione in tempi brevi, l'applicazione di criteri per indirizzare la ricerca in una direzione convergente verso quella ottimale. Si è soliti implementare i suddetti criteri tramite delle funzioni che associano a ciascun nodo dell'albero un valore di stima della distanza della configurazione del nodo in questione dalla soluzione; tali funzioni sono chiamate *euristiche*. Gli algoritmi che sfruttano le euristiche per indirizzare la ricerca verso il goal sono chiamati, appunto, *euristici*, e sono tipicamente più veloci dei tradizionali metodi di ricerca in profondità ed ampiezza, chiamati algoritmi *ciechi*, dato che tutte le configurazioni sono trattate allo stesso pari.

In questa tesina, ci occupiamo di scrivere un programma che ci consenta di confrontare le performance di alcuni tra i più famosi algoritmi di ricerca: in particolare analizzeremo *BFS* e *DFS* come algoritmi ciechi ed *A** e *IdA** come algoritmi informati; nel caso degli algoritmi informati, vedremo anche come la scelta di una funzione euristica ottimale influisca sulle performance globali: le euristiche che andremo ad utilizzare sono *Tfp* e *Manhattan*.

2 Il Gioco dell'8

Il problema di ricerca che prendiamo sotto esame in questa tesina è il *gioco dell'8*, semplificazione del più famoso *gioco del 15*. Il gioco consiste nell'ordinare una serie di tessere numerate, da 1 a 8, inserite all'interno di una struttura a matrice 3x3, che permette il movimento orizzontale e verticale di ogni tessera, sfruttando la casella rimasta vuota. Una volta scelta la configurazione *goal*, lo scopo del gioco è trovare la serie di mosse che porta la configurazione iniziale al *goal* stesso.

Per come il gioco è strutturato, non è possibile raggiungere sempre il *goal* da qualunque configurazione iniziale: vi sono infatti due *orbite* distinte di configurazioni, non comunicanti, che corrispondono a due modi di incastrare fisicamente le tessere nella matrice. Data quindi una configurazione iniziale, è possibile raggiungere soltanto configurazioni appartenenti alla stessa orbita.

Sebbene vi siano $9!$ configurazioni possibili, il problema riguarda quindi soltanto metà di esse alla volta: gli algoritmi dovranno dunque esplorare al più $\frac{9!}{2} = 181440$ configurazioni.

	4	3
2	8	6
1	7	5

1	2	3
4	5	6
7	8	

Figura 1: stato iniziale e stato goal

3 Struttura del programma

Il software sviluppato si fonda su quattro passi fondamentali:

1. Generazione degli stati iniziali
2. Calcolo dell'orbita
3. Esecuzione dell'algoritmo
4. Report statistico

Per la stesura del programma abbiamo scelto di utilizzare il linguaggio di programmazione *Ruby*. I sorgenti del software sono consultabili online¹ e sono rilasciati sotto licenza *GPL 3*.

3.1 Generazione degli stati iniziali

Nella prima fase il software genera degli stati iniziali (rappresentati da una stringa casuale di 9 caratteri univoci nell'intervallo 0-8 e in cui il carattere 0 rappresenta la casella vuota) che vengono poi elaborati nelle fasi successive. Il numero di configurazioni da creare viene passato al software dall'utente al momento dell'esecuzione.

3.2 Calcolo dell'orbita

Il calcolo dell'orbita avviene scorrendo da sinistra verso destra la stringa rappresentante la configurazione e contando il numero di valori minori del carattere in esame (escludendo lo 0 corrispondente alla casella vuota); al termine del calcolo, la somma di tali valori determina se la configurazione appartiene ad un'orbita pari o dispari. Per poter calcolare la soluzione per tutte le possibili configurazioni iniziali, abbiamo scelto due goal diversi per orbite pari e dispari, rappresentati in figura 2.

1	2	3
4	5	6
7	8	

1	2	3
4	5	6
8	7	

Figura 2: goal per orbite pari e dispari

¹ <http://www.lilik.it/~anarki/IA/otto.tgz>

3.3 Algoritmi

Abbiamo provveduto a scrivere delle implementazioni per ciascuno degli algoritmi menzionati nell'introduzione, avvalendoci delle classi della libreria standard di Ruby, ad eccezione di due pacchetti, *ruby-tree* e *PriorityQueue*; nel caso di *ruby-tree*, abbiamo effettuato alcune ottimizzazioni del codice disponibile per garantire al programma la massima velocità di esecuzione. Inoltre, per ciascun algoritmo abbiamo utilizzato le seguenti strutture:

- una tabella *hash* per manenere in cache la lista dei nodi dell'albero già visitati
- il tempo di sistema, per tenere traccia della durata di ciascuna esecuzione
- una classe *Stats* per controllare e stampare agevolmente le statistiche provenienti dall'esecuzione

3.3.1 Breadth-First Search

Questo algoritmo effettua la ricerca in ampiezza, ovvero esplora tutti i nodi dello stesso livello prima di scendere a quello successivo. Tale meccanismo garantisce *completezza* in quanto vengono analizzati tutti i nodi della stessa orbita, e *ammissibilità*, poiché la soluzione eventualmente trovata, è quella ad altezza minore.

Internamente l'algoritmo utilizza una coda di tipo *FIFO* per memorizzare i nodi da espandere; nella nostra implementazione abbiamo invertito l'utilizzo standard della coda, inserendo in coda e prelevando dalla testa, per la migliore performance nell'implementazione della libreria standard di Ruby di tali metodi.

```
while !@queue.empty?
  node = @queue.pop
  if @lists[node.name] != 1
    if node.name == @goal
      return true
    else
      node.expand
      @lists[node.name] = 1
      node.children do |child|
        @queue.unshift(child)
      end
    end
  else
    node.parent.remove!(node)
  end
end
```

I nodi espansi, a livello $d - 1$, sono $1 + b + b^2 + \dots + b^{d-1} = \frac{b^d - 1}{b - 1}$. Se supponiamo di trovare la soluzione a metà del livello d , alla formula precedente, dobbiamo sommare $\frac{1 + b^d}{2}$. Quindi il numero totale dei nodi espansi è pari a $\left(\sum_{j=0}^{d-1} b^j \right) + \frac{b^d + 1}{2} = \frac{b^d - 1}{b - 1} + \frac{b^d + 1}{2} \approx \frac{b^d}{2} \left(1 + \frac{1}{b} \right)$, dove b è il fattore di ramificazione e d è la profondità dell'albero.

3.3.2 Depth-First Search

Questo algoritmo, a differenza di *BFS*, agisce in profondità, ovvero espande sempre tutti i figli sinistri dei nodi, fino al termine dell'albero, quando poi risale di un livello e prosegue verso destra.

L'algoritmo è *completo*, ma non *ammissibile*, in quanto non è detto che il *goal* trovato sia quello a minore profondità. Internamente, i nodi espansi sono mantenuti dentro uno stack.

```
while !@queue.empty?
  node = @queue.pop
  if @lists[node.name] != 1
    if node.name == @goal
      return true
    else
      node.expand
      @lists[node.name] = 1
      node.children do |child|
        @queue.push(child)
      end
    end
  else
    node.parent.remove!(node)
  end
end
```

Il numero di nodi espansi oscilla tra $(d+1)$ e $\left(\sum_{j=0}^d b^j = \frac{b^{d+1}-1}{b-1}\right)$, quindi il numero medio di nodi espansi è pari a:

$$\frac{(d+1) + \frac{b^{d+1}-1}{b-1}}{2} \approx \frac{b^d}{2}$$

Possiamo osservare che la complessità di *DFS* è minore di quella di *BFS* di un fattore $\left(1 + \frac{1}{b}\right)$; è quindi conveniente usare *DFS* per fattori di ramificazione piccoli.

3.3.3 A*

Si tratta della versione *ammissibile* dell'algoritmo A. E' un algoritmo di ricerca informato, poiché utilizza una funzione in grado di scegliere il nodo da espandere; questa è calcolata come $f(n) = g(n) + h'(n)$, dove $g(n)$ è l'altezza del nodo e $h'(n)$ è un'euristica scelta dal chiamante dell'algoritmo, che sottostima la distanza del nodo in questione dal *goal*.

```
while !@queue.empty?
  node = @queue.delete_min_return_key
  if @lists[node.name] != 1
    if node.name == @goal
      return true
    else

```

```

node.expand(@goal)
@lists[node.name] = 1
node.children do |child|
  @queue[child] = child.getEuristicVal
end
end
else
node.parent.remove!(node)
end
end
end

```

Per il mantenimento dei nodi, abbiamo utilizzato un *min-heap*, fornito dalla classe *PriorityQueue*, struttura che mantiene sempre in radice l'elemento con il minimo valore euristico, e sul quale è possibile effettuare operazioni con complessità $\Theta(\log_2(n))$.

3.3.4 *IdA**

Iterative Deepening A* è un algoritmo informato che utilizza l'euristica per determinare la potatura dell'albero di ricerca. Ad ogni iterazione si espandono soltanto i nodi con euristica minore della soglia di potatura, che viene poi incrementata se l'iterazione non dà esito positivo. Internamente, i nodi vengono mantenuti in uno stack.

```

# c è la soglia di potatura
c = 1
while c < 181440
  @queue.push(@root)
  while !@queue.empty?
    node = @queue.pop
    if node.name == @goal
      return true
    else
      node.expand(@goal)
      node.children do |child|
        priority = child.getEuristicVal
        if priority <= c
          @queue.push(child)
        end
      end
    end
  end
  c += 1
end
end

```

3.4 Euristiche

Entrambe le euristiche utilizzate rispettano il vincolo $h'(n) \leq h(n)$, dove $h(n)$ rappresenta la distanza del nodo dal goal. Questa è la condizione necessaria affinché gli algoritmi siano *ammissibili*, ovvero convergano al *goal* più vicino.

3.4.1 Tessere Fuori Posto

L'euristica calcola il numero di tessere fuori posto rispetto alla condizione di *goal*. Nonostante sia semplice concettualmente e facile da calcolare, l'euristica non è molto rappresentativa, in quanto rappresenta le $\frac{9!}{2}$ possibilità soltanto in 9 gruppi distinti. Non tiene conto, quindi, del numero di mosse che servono per riportare la tessera a posto, ma solo del fatto che non è nella sua casella.

3.4.2 Manhattan

Questa euristica utilizza la *geometria del Taxi* per calcolare la distanza tra la configurazione e il *goal*. In particolare, effettua la somma delle distanze *Manhattan* di ciascuna tessera dalla sua posizione nella configurazione *goal* mediante la formula $M(P_1, P_2) = |x_1 - x_2| + |y_1 - y_2|$, dove P_1, P_2 rappresentano le posizioni della tessera rispettivamente nella configurazione del nodo e in quella del *goal*.

L'euristica, sebbene sia piuttosto precisa e molto performante, non tiene conto però della difficoltà di scambiare due tessere adiacenti.

4 Analisi delle prestazioni

Per valutare le performance di ciascun algoritmo utilizzeremo, oltre al tempo di esecuzione, troppo legato alla bontà delle implementazioni delle strutture dati utilizzate e alla velocità dell'hardware, altri parametri specifici del problema.

4.1 Penetranza

La penetranza, che mette in relazione la lunghezza L del cammino generato con il numero T di nodi espansi, è calcolata come:

$$P = \frac{L}{T} \quad \text{con} \quad 0 \leq P \leq 1$$

Un algoritmo è tanto migliore, quanto più P si avvicina ad 1; il valore 1 starebbe a significare che ogni nodo espanso dall'algoritmo è appartenente al cammino ottimo.

4.2 Fattore di ramificazione

Il fattore di ramificazione B rappresenta il numero medio di successori di ogni nodo, e si calcola come

$$T = \frac{B(B^L - 1)}{B - 1}$$

dove T e L sono intesi come nel precedente paragrafo. Essendo la funzione non invertibile, è necessario utilizzare un'algoritmo numerico come quello di bisezione per procedere alla sua stima. Tanto più il fattore di ramificazione è basso, tanto meglio si comporta l'algoritmo, evitando di espandere nodi lungo percorsi che non portano verso il *goal*.

Di seguito il programma *Ruby* usato per calcolare numericamente il parametro.

```

def find(l,t)
    bound = [0.01, 2.00]
    mid = (bound[1] + bound[0]) / 2
    res = ((mid*(mid**l-1))/(mid-1))-t
    while res.abs > 0.01
        if res > 0
            bound[1] = mid
        else
            bound[0] = mid
        end
        mid = (bound[1] + bound[0]) / 2
        res = ((mid*(mid**l-1))/(mid-1))-t
    end
    return mid
end

```

5 Esecuzione del programma

Il programma è stato fatto girare su due dataset: il primo contenente 100 configurazioni iniziali, è stato testato su tutti gli algoritmi, mentre il secondo, da 10000 configurazioni, è stato usato soltanto con A^* e IdA^* con euristica *Manhattan*. Nelle tabelle 1 e 2 è possibile visualizzare i risultati dei test. La colonna *Mem* è indicativa dell'occupazione di memoria dinamica derivante dall'esecuzione dell'algoritmo ed identifica il numero massimo di nodi presenti nella coda di appoggio. Tutti i dati sono valori medi sulle configurazioni iniziali testate.

	Tempo	L	T	P	B	Mem
BFS	18,583	22,55	126229,67	$2,56 \cdot 10^{-4}$	1,61	31916,58
DFS	7,324	66149,61	121838,23	$5483,19 \cdot 10^{-4}$	1,00	46543,16
A*/Tfp	3,703	22,55	24675,54	$35,64 \cdot 10^{-4}$	1,49	8329,69
A*/Man	0,527	22,59	3620,15	$203,98 \cdot 10^{-4}$	1,35	1360,51
IdA*/Tfp	13,812	22,55	94953,92	$22,38 \cdot 10^{-4}$	1,59	16,48
IdA*/Man	0,924	22,57	6653,03	$163,27 \cdot 10^{-4}$	1,40	13,78

Tabella 1: risultati del test su 100 configurazioni

	Tempo	L	T	P	B	Mem
A*/Man	0,412	21,97	2880,31	$252,17 \cdot 10^{-4}$	1,35	1087,10
IdA*/Man	0,633	21,94	5186,03	$212,03 \cdot 10^{-4}$	1,39	13,42

Tabella 2: euristiche Manhattan a confronto (10000 configurazioni)

6 Conclusioni

Possiamo subito notare che le prestazioni degli algoritmi informati (A^* e IdA^*), sono sicuramente migliori rispetto a quelle degli algoritmi ciechi, fatta eccezione per IdA^*/Tfp , che ottiene risultati non troppo distanti da BFS , con tutta probabilità a causa dell'euristica Tfp non ottimale, come discusso nel paragrafo 3.4.1.

Si nota anche che DFS risulta avere un valore di penetranza decisamente migliore rispetto a tutti gli altri algoritmi: questo è previsto, considerando che DFS , per definizione, espande cammini molto lunghi in profondità. Il valore è però ingannevole dato che si tratta di un algoritmo non *ammissibile*. Da notare invece che, essendo tutti gli altri algoritmi *ammissibili*, ed essendo tutti i rami dell'albero a costo unitario, essi sono anche *ottimi*. Sempre per quanto riguarda la penetranza, sono invece da mettere in rilievo le ottime prestazioni degli algoritmi informati se usati con la euristica *Manhattan*, che risulta tagliare lo spazio di ricerca in modo decisamente più efficiente rispetto a Tfp . Come previsto è invece molto basso il valore del parametro per BFS poiché esso espande un numero di nodi molto grande. I valori del fattore di ramificazione ricalcano l'andamento della penetranza, con buoni indici per gli algoritmi che usano l'euristica *Manhattan*.

Interessante da sottolineare è anche il diverso comportamento di A^*/Man e IdA^*/Man per quanto riguarda l'occupazione dinamica di memoria: si nota come l'*Iterative Deepening*, pur lievemente penalizzato da punto di vista prestazionale, ha un'occupazione di memoria di due ordini di grandezza inferiore rispetto all'algoritmo A . Consigliamo quindi il suo uso nelle situazioni in cui la quantità di memoria utilizzata ha un costo superiore rispetto a quello dei cicli di clock del processore. In tutti gli altri casi, consigliamo invece l'uso dell'algoritmo A^* con euristica *Manhattan*, in quanto è quello con risultati complessivamente migliori.