Risoluzione del Sudoku: due approcci.

Relazione a cura di Marco Cornolti del progetto "Sudoku",
corso di Laboratorio di Strutture Dati,
proff Chiara Bodei e Francesco Romani, A.A. 2005/2006

Introduzione

Il Sudoku è un gioco matematico piuttosto semplice, nel senso che è definito da poche regole e le mosse ragionevoli necessarie a risolvere un'istanza non derivano da un ragionamento complicato. Se negli scacchi un giocatore può vincere contro un altro in moltissimi modi diversi, nel sudoku la combinazione vincente per riempire le caselle vuote è una sola, e tutte le altre combinazioni sono sbagliate.

Probabilmente esistono diversi algoritmi per risolvere il sudoku, ma in questa relazione esporrò il funzionamento di due in particolare che funzionano in modo piuttosto diverso tra loro. Il primo funziona con la ricerca a spazi di stati, mentre il secondo emula lo schema di pensiero umano.

La ricerca a spazi di stati

Il primo metodo esposto funziona con un procedimento piuttosto potente ma difficilmente attuabile da una mente umana, per la grande quantità di informazioni processate logicamente in parallelo.

Il metodo main sfrutta due package:

1. `engine`: un motore che effettua la ricerca a spazi di stati costruito per risolvere in generale i giochi ad un giocatore
2. `sudoku`: l'implementazione del Sudoku.

Entrambi i package sono forniti già scritti. Per far funzionare il motore di ricerca a spazi di stati `engine` e l'implementazione del gioco `sudoku` è stato necessario scrivere un terzo package, `sudoku_resolver`, che con la classe `SudokuState`, che rappresenta uno stato nell'evoluzione del gioco, fornisce i metodi necessari al motore per funzionare col package `sudoku`. Inoltre è stata scritta la classe col metodo `main()`, chiamata `SudokuSearch` che si occupa di risolvere tutti i giochi presenti in `SudokuGames` uno dopo l'altro usando il motore e aggiornare le statistiche dei tempi.

Il funzionamento dell'algoritmo prevede un ciclo di passaggi che porta alla soluzione del problema iniziale o all'abbandono per l'impossibilità di risolverlo. I passaggi sono i seguenti:

1. Dall'istanza di partenza viene creato uno stato del gioco
2. Vengono applicate le regole di semplificazione sullo stato
3. Si espande lo stato negli stati derivati dalle possibili mosse (occupazione delle caselle) e per ogni stato trovato si calcola la funzione di valutazione. Lo stato viene aggiunto alla coda solo se la funzione di valutazione ha un valore inferiore a una costante fissata.
4. Se uno degli stati raggiunti è la soluzione, l'algoritmo si ferma, altrimenti si ritorna ad eseguire il punto 2 sullo stato che ha funzione di valutazione più bassa tra quelli nella coda. Se non ci sono stati nella coda, l'algoritmo termina con la non-soluzione.

La risoluzione “umana”

Un altro approccio per la creazione di un algoritmo di risoluzione del sudoku è quello più vicino al modo di operare degli umani. Proprio perché il Sudoku è un gioco molto semplice, è possibile risolvere una sua istanza applicando poche regole un gran numero di volte fino al raggiungimento della soluzione. Il package `sudoku_human` è stato completamente sviluppato da me.

Per trovare le regole è stato sufficiente giocare una partita con la matita e pensare a come formalizzarle per farle eseguire dalla macchina. Per le correzioni successive dell'algoritmo ho analizzato le situazioni di stallo in cui si trovava il programma e ho trovato le soluzioni.

Innanzitutto, invece di pensare la griglia del Sudoku come composta da 9x9 numeri, è necessario pensarla come una griglia di 9x9 caselle, oggetti java (definiti dalla classe `Caselle`) che rappresentano le *possibilità*, ovvero quei numeri che la casella può ancora assumere. La creazione di questo oggetto è necessaria perché ad ogni iterazione, le regole non definiscono quasi mai l'esatto valore di una casella, ma piuttosto tolgono delle possibilità alle caselle, e il loro valore viene definito per esclusione. All'inizio dell'esecuzione dell'algoritmo, vi sono alcune caselle che hanno una sola possibilità vera (quelle che definiscono lo stato iniziale del problema) e tutte le altre hanno tutte le possibilità vere, ovvero sono completamente indefinite. Durante l'esecuzione dell'algoritmo, le possibilità verranno eliminate ad ogni iterazione e al termine dell'algoritmo tutte le caselle avranno un'unica possibilità, ovvero tutte saranno completamente definite (in caso di successo dell'algoritmo).

Le regole utilizzate sono simili a quelle esposte nei complementi sul Sudoku, anche se implementate in maniera diversa per ragioni di ottimizzazione:

A. Per ogni casella C_i non definita, si controllano tutte le possibilità che hanno le altre caselle $C_1 ... C_{i-1} ... C_{i+1} ... C_9$ nella stessa riga. Se la casella in esame è l'unica che può assumere un certo valore, verrà definita con questo valore. Analogamente si procede per le colonne e per i blocchi.

B. Per ogni casella definita con un numero *n*, tutte le altre caselle della stessa riga, blocco e colonna non possono avere tra le possibilità *n*.

L'esecuzione dell'algoritmo con l'uso di queste due regole garantisce un veloce svolgimento (sui 5 msec) della maggioranza dei problemi difficili forniti nella collezioni di `SudokuGames`. Tuttavia alcuni problemi particolarmente difficili non potevano essere risolti, in quanto si raggiungevano degli stati in cui era impossibile eliminare altre possibilità. Questi problemi erano, in numero, 202 sui 450 di `SudokuGames`.

Per risolvere questi problemi ho analizzato alcuni di questi stati, e implementato una terza regola.

La regola C

Nella risoluzione del gioco 55, l'algoritmo si bloccava in questo stato (in ogni casella sono scritte le possibilità non ancora escluse):

248 9	369	236 8	1	49	7	468	5	69
4578 9	1567 9	678	2	459	68	468	3	167 9
4578 9	135 679	3678	36	459	368	468	2	167 9
25	35	1	4	8	36	7	9	26
6	4	9	7	2	5	3	1	8
278	37	2378	36	1	9	5	4	26
3	2	4	8	6	1	9	7	5
79	679	67	5	3	2	1	8	4
1	8	5	9	7	4	2	6	3

dopo aver osservato per un po' la griglia ci si rende conto che esistono ancora delle possibilità da eliminare. Nell'esempio, il blocco (3,2), ovvero quello con sfondo azzurro, contiene solo due caselle non definite, che possono entrambe assumere i valori 2 e 6. È impossibile, per ora, determinare quale delle due assumerà, per esempio, il valore 6, ma sappiamo che una delle due sicuramente lo assumerà. Per questo possiamo negare la possibilità che altre caselle della colonna, all'infuori delle due nel blocco prese in esame, assumano i valori 6 e 2. Nell'esempio questa regola toglie la possibilità 6 nelle caselle (9,1), (9,2) e (9,3).

Questo è il risultato dell'esecuzione della regola C sullo stato nella figura precedente.

248 9	369	236 8	1	49	7	468	5	9
4578 9	1567 9	678	2	459	68	468	3	179
4578 9	1567 9	678	36	459	368	468	2	179
25	35	1	4	8	36	7	9	26
6	4	9	7	2	5	3	1	8
278	37	2378	36	1	9	5	4	26
3	2	4	8	6	1	9	7	5
79	679	67	5	3	2	1	8	4
1	8	5	9	7	4	2	6	3

Più generalmente, la regola dice che:

C. Se è possibile accertare che una possibilità è isolata in una riga (o colonna) e all'interno di un blocco, sarà possibile negare la possibilità che lo stesso valore sia assunto dalle caselle della stessa riga (o colonna) all'infuori del blocco.

Questa regola ovviamente ha scarsa probabilità di essere applicata (non è frequente una situazione come quella dell'esempio), mentre ha lo stesso costo in tempo dell'applicazione delle altre regole. Per questo l'algoritmo principale la usa, ad ogni passaggio, solo se l'applicazione delle prime due regole A e B non ha avuto alcun effetto sull'istanza.

Con l'utilizzo della regola C, i problemi irrisolvibili sono scesi da 202 a 100.

I tentativi

Nonostante la creazione della regola C permetta la risoluzione di 102 problemi in più che non erano risolvibili con le sole regole A e B, restano 100 problemi che raggiungono degli stati non risolvibili in cui le tre regole non hanno più effetto, ad esempio quello nella tabella seguente, raggiunto durante la risoluzione del problema 192:

4	6	3	8	1	579	59	57	2
7	1	9	3	2	56	4	8	56
2	5	8	4	67	679	169	17	3
8	3	6	7	4	2	15	15	9
5	9	2	1	68	68	7	3	4
1	7	4	9	5	3	2	6	8
9	2	1	6	3	78	58	4	57
6	8	7	5	9	4	3	2	1
3	4	5	2	78	1	68	9	67

Visto che nella maggior parte dei casi, questi stadi avevano poche caselle non ancora definite e comunque con poche possibilità, ho scelto l'approccio della forza bruta, ovvero la definizione delle caselle indefinite scegliendo tra le possibilità in maniera sostanzialmente casuale. Questo garantiva ovviamente la soluzione di tutti i problemi, ma mentre la stragrande maggioranza veniva risolta in tempi molto bassi, inferiori ai 50 millesimi di secondo, altri avevano un tempo di risoluzione decisamente inaccettabile. 7 ore e mezzo, in alcuni casi.

La coda degli stati

L'inefficienza del metodo precedente è causata dal fatto che le permutazioni con cui riempire le caselle non definite restano in alcuni casi, al contrario di quanto suggerisce l'apparenza, una quantità enorme. Nell'esempio sono 1.769.472 ma in alcuni casi raggiungono ordini di grandezza ben più grandi: nel problema n. 324 sono $4{,}6 \cdot 10^{37}$. Per questo, una volta imboccata una strada (definita una casella come una qualsiasi delle sue possibilità), la soluzione può giungere subito se la possibilità scelta è quella giusta, mentre l'algoritmo lavora inutilmente per delle ore prima di rendersi conto che la scelta è sbagliata.

Per risolvere questa situazione ho implementato una coda FIFO in cui vengono inserite, nel caso che le normali regole non abbiano effetto su una data istanza, delle istanze del problema semplificate. Queste istanze sono ottenute prendendo in esame una casella non definita *c* (con possibilità $p_0 \cdots p_{n-1}$) e aggiungendo alla coda le istanze con quella casella definita in tutti i modi possibili, ovvero *n* istanze ognuna con il valore p_i assegnato a *c*. Naturalmente una sola delle griglie aggiunte ad ogni passaggio è giusta mentre tutte le altre hanno la possibilità assegnata in modo errato e verranno scartate appena l'algoritmo se ne renderà conto.

Ad ogni istanza a partire dall'inizio della coda vengono applicate le tre regole finché non si raggiunge una soluzione oppure le regole non hanno più effetto. Nel secondo caso, se le regole non modificano lo stato, l'istanza viene espansa nuovamente e le istanze generate vengono inserite in fondo alla coda FIFO. In questo modo vengono prese in esame le istanze nell'ordine in cui sono generate, e tra le prime *n*, sicuramente una avrà la possibilità determinata in maniera esatta.

Per semplificare la comprensione riporto schematicamente i passaggi effettuati dall'algoritmo umano globalmente:

1. Si aggiunge alla coda FIFO delle istanze da risolvere quella di base.
2. Se non ci sono istanze nella coda, si termina con la non-soluzione. Altrimenti si prende in esame la prima istanza della coda e vi si applicano le regole A e B.
3. Se le regole A e B non hanno modificato lo stato, si applica la regola C.
4. Se siamo giunti alla soluzione, si termina.
5. Se non sono più rispettate tutte le leggi del sudoku (ci sono caselle con zero possibilità) significa che una casella è stata definita male oppure che non è possibile risolvere l'istanza di partenza. Si passa dunque al punto 8, ovvero si cancella l'istanza in esame e si continua con l'analisi delle altre istanze nella coda (se ve ne sono).
6. Se almeno una delle tre regole ha modificato lo stato, si torna al punto 2.
7. Se, al contrario, le tre regole non hanno modificato lo stato, si procede con l'espansione: si sceglie una casella non definita e si aggiungono alla coda le istanze con le possibilità al posto della casella indefinita.
8. Si cancella dalla coda la prima istanza (quella attualmente in esame) e si ritorna al punto 2.

Qual è l'algoritmo migliore?

Per confrontare la potenza dei due algoritmi, quello a ricerca di spazi di stati e quello “umano”, ho creato una classe `Statistiche` con i metodi per valutare la velocità degli algoritmi nel risolvere i problemi del sudoku. Questi metodi semplicemente archiviano durante l'esecuzione i tempi di risoluzione delle istanze e prima che termini il programma stampano il riassunto dei dati.

I test sono stati fatti tenendo più libero possibile il processore, ed hanno dato risultati simili tutte le volte che sono stati eseguiti. La macchina usata ha come CPU un Intel Centrino a 1.60 Mhz con RAM a 400 Mhz, il sistema operativo usato è linux con kernel 2.6.16 ed i programmi sono stati eseguiti sia in ambiente Eclipse che da shell, dando simili risultati.

Ecco dunque i risultati per la ricerca a spazi di stati:

```
 --- Statistiche sulle soluzioni ---
Problemi iniziati: 450
Problemi risolti: 450
Problemi non risolti: 0

 --- Statistiche sui tempi ---
Tempo globale impiegato: 2853
Media tempo impiegato: 6.34
Varianza tempo impiegato: 133.3577333333321
Massimo tempo impiegato: 133 nell'istanza n.143
Tempo impiegato per risolvere l'istanza col massimo di espansioni (130): 129

 --- Statistiche sulle espansioni ---
Globale espansioni: 44910
Media espansioni: 99.8
Massimo numero di espansioni in un'istanza: 2016 nell'istanza n.130
Numero di espansioni nell'istanza col tempo massimo (143): 1847

 --- Classifiche sui tempi - primi 10 ---
Posizione 143 tempo: 133msec
Posizione 130 tempo: 129msec
Posizione 142 tempo: 104msec
Posizione 131 tempo: 67msec
Posizione 127 tempo: 60msec
Posizione 90 tempo: 46msec
```

```
Posizione 444 tempo: 40msec
Posizione 334 tempo: 38msec
Posizione 324 tempo: 31msec
Posizione 97 tempo: 30msec

 --- Classifiche sui tempi - ultimi 10 ---
Posizione 326 tempo: 2msec
Posizione 160 tempo: 2msec
Posizione 375 tempo: 2msec
Posizione 169 tempo: 2msec
Posizione 228 tempo: 2msec
Posizione 378 tempo: 2msec
Posizione 449 tempo: 3msec
Posizione 133 tempo: 3msec
Posizione 148 tempo: 3msec
Posizione 446 tempo: 3msec
```

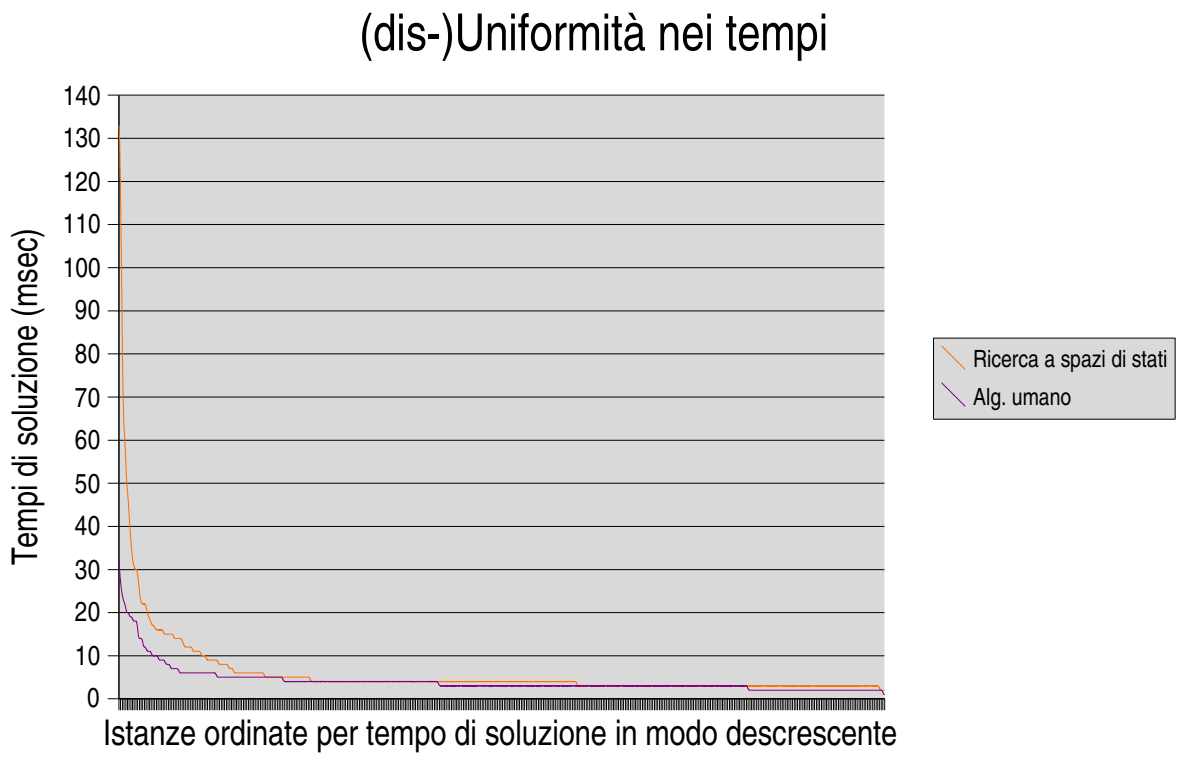
Mentre questi sono i risultati per il metodo “umano”:

```
 --- Statistiche sulle soluzioni ---
Problemi iniziati: 450
Problemi risolti: 450
Problemi non risolti: 0

 --- Statistiche sui tempi ---
Tempo globale impiegato: 1911
Media tempo impiegato: 4.246666666666667
Varianza tempo impiegato: 12.541377777777774
Massimo tempo impiegato: 31 nell'istanza n.190
Tempo impiegato per risolvere l'istanza col massimo di espansioni (190): 31

 --- Statistiche sulle espansioni ---
Globale espansioni: 446
Media espansioni: 0.9911111111111112
Massimo numero di espansioni in un'istanza: 22 nell'istanza n.190
Numero di espansioni nell'istanza col tempo massimo (190): 22

 --- Classifiche sui tempi - primi 10 ---
Posizione 190 tempo: 31msec
Posizione 23 tempo: 28msec
Posizione 188 tempo: 25msec
Posizione 141 tempo: 23msec
Posizione 142 tempo: 22msec
Posizione 187 tempo: 22msec
Posizione 359 tempo: 19msec
Posizione 117 tempo: 18msec
Posizione 120 tempo: 18msec
Posizione 324 tempo: 18msec

 --- Classifiche sui tempi - ultimi 10 ---
Posizione 218 tempo: 1msec
Posizione 284 tempo: 1msec
Posizione 46 tempo: 2msec
Posizione 448 tempo: 2msec
Posizione 208 tempo: 2msec
Posizione 446 tempo: 2msec
Posizione 173 tempo: 2msec
Posizione 22 tempo: 2msec
Posizione 36 tempo: 2msec
Posizione 442 tempo: 2msec
```

Il dato più significativo, ovvero la somma dei tempi di risoluzione di tutte le istanze di

`SudokuGames`, assegna la medaglia d'oro all'algoritmo umano, che ha bisogno in media di 2/3 del tempo di cui ha bisogno l'altro algoritmo.

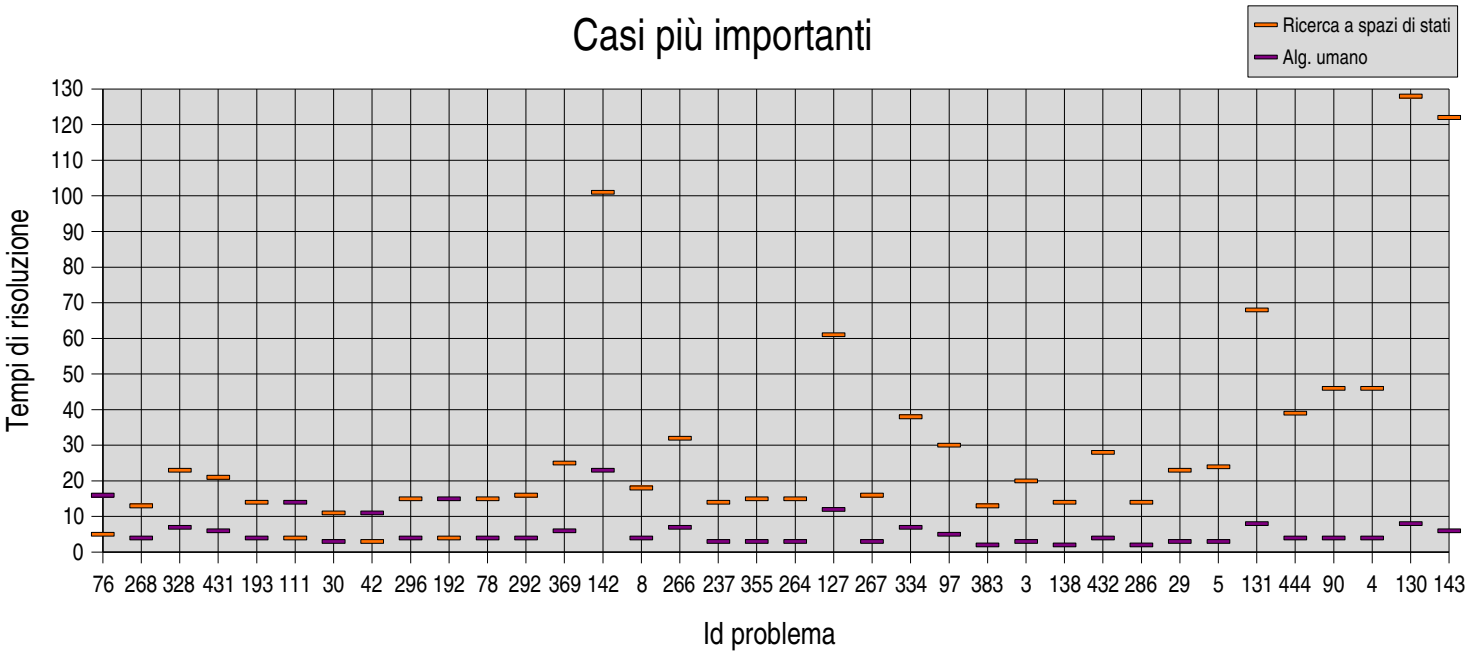
Il confronto tra il numero di espansioni non ha senso, visto che per l'algoritmo che usa la ricerca a spazi di stati, quello delle espansioni è il metodo basilare, mentre in quello umano è solo l'asso nella manica. Ma studiamo più in dettaglio gli altri risultati. Sempre riguardo ai tempi, come indica la varianza (133 vs. 12), l'algoritmo umano ha tempi di soluzione più uniformi rispetto all'altro. Infatti osservando i tempi nella classifica delle 10 risoluzioni più lente, quello al primo posto (133 msec) è di ben 5 volte superiore a quello al decimo posto (30 msec), mentre nell'algoritmo



umano è meno del doppio.

Nel grafico sopra sono riportati i tempi di risoluzione ordinati in modo decrescente per ogni algoritmo. Questo significa che, ad esempio, la linea viola per l'istanza più a sinistra indica il tempo di risoluzione del problema più lento per l'algoritmo umano, mentre la linea arancione indica il tempo più lento per l'algoritmo di ricerca a spazi di stati, ma questi due problemi non sono necessariamente gli stessi.

Nel seguente grafico si confrontano sempre i tempi di risoluzione tra i due algoritmi, ma sono selezionati i casi in cui il tempo più veloce è almeno 3 volte superiore rispetto all'altro e la loro differenza è di almeno 4 millisecondi. Questo per restringere l'analisi ai 36 casi più significativi. I valori sono ordinati per il rapporto tra il tempo maggiore e il tempo minore.



Come vediamo dal grafico, dei 36 casi più significativi, in 4 risulta più veloce l'algoritmo di ricerca a spazi di stati, e questi 4 sono tra i primi 10 più a sinistra, ovvero sono meno significativi, mentre negli altri 32 l'algoritmo più veloce è quello umano, che nel caso estremo è di 20 volte più veloce (6 vs. 122 nell'istanza 143).

Tuttavia possiamo trovare delle linee comuni tra i comportamenti dei due algoritmi, infatti, pur essendo questi strutturati in maniera molto diversa, vi sono due problemi, il 142 e il 324, che sembrano essere in assoluto i più difficili, visto che compaiono nelle classifiche dei più lenti per entrambi gli algoritmi. In particolare sono classificati rispettivamente come 5° e 10° nell'algoritmo umano, mentre sono 3° e 8° nell'algoritmo di ricerca a spazi di stati.

9 agosto 2006,
Marco Cornolti