Relazione progetto A.I.

Federico Nespoli - 1035120

Sommario

INTRODUZIONE

I temi trattati sono il gioco del 15, quello dell'8 e l'implementazione di diversi metodi risolutivi.

Il gioco consiste in una tabellina di forma quadrata, composta da 3 righe e 3 colonne (quindi 9 posizioni per il gioco dell'8), oppure divisa in quattro righe e quattro colonne (quindi 16 posizioni, per il gioco del 15), su cui sono posizionate tessere quadrate, numerate progressivamente a partire da 1. Le tessere possono scorrere in orizzontale o verticale, ma il loro spostamento è ovviamente limitato dall'esistenza di un singolo spazio vuoto. Lo scopo del gioco è riordinare le tessere dopo averle "mescolate" in modo casuale; in particolare, la posizione da raggiungere è quella con il numero 1 in alto a sinistra e gli altri numeri a seguire da sinistra a destra e dall'alto in basso fino al 15, seguito dalla casella vuota.

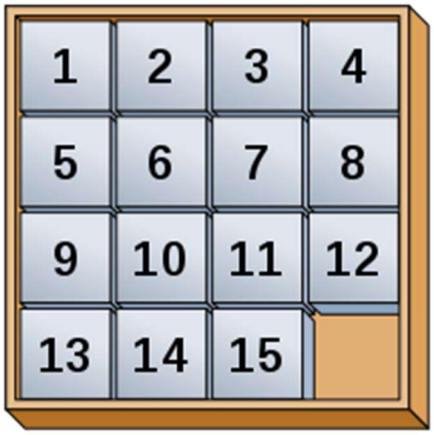

Figura 1- Gioco del 15

L'applicazione realizzata permette di risolvere (per quanto possibile) il gioco in questione con vari metodi risolutivi di ricerca informata e non informata, in modo da poterne confrontare le prestazioni e studiarne il comportamento.

FORMULAZIONE DEL PROBLEMA

- Insieme degli stati: Combinazioni delle tessere contenute nella tabella
- Stato iniziale: Una delle possibili combinazioni delle celle contenute nella tabella
- Azioni ammissibili: Scambiare la cella vuota con una delle celle confinanti
- Modello di transizione: Deterministico
- Test obiettivo: Tutte le celle ordinate e la cella vuota in basso a destra
- Costo passo: tutte le mosse hanno lo stesso costo (1)

ANALISI TEORICA DEI METODI RISOLUTIVI

Per risolvere questo problema, sono stati applicati svariati metodi risolutivi, che verranno elencati e introdotti in seguito.

Ricerca non informata

La ricerca non informata è una strategia di ricerca per trovare una o più soluzioni a un problema. È una strategia utilizzata mediante lo sviluppo di appositi algoritmi in grado di ricercare la soluzione grazie l'espansione combinatoria degli stati.

È definita "non informata" perché non si serve/dispone di alcun tipo di conoscenza relativa al problema, essendo perciò, nella maggior parte dei casi, non efficiente.

I parametri che ci permettono di analizzare teoricamente questi algoritmi sono 3:

1. b: Branching factor;
2. d: Profondità;
3. m: Lunghezza massima dei cammini.

Ricerca in Profondità:

La ricerca in profondità è un tipo di ricerca non informata che espande i nodi a partire dai più profondi dell'albero di ricerca.

Analisi ricerca in profondità:

- Completezza: in questo caso, no;
- Ottimalità: no;
- Complessità temporale: O(b^m);
- Complessità spaziale: O(bm).

Questo algoritmo è molto semplice, infatti può soffrire di eventuali problemi riguardanti i loop nella ricerca su albero.

L'algoritmo di ricerca è implementabile attraverso una coda LIFO (Last Input First Output) e può essere realizzato anche in maniera ricorsiva.

Ricerca in Ampiezza

La ricerca in ampiezza è un tipo di ricerca che consiste nell'espandere un nodo, solo nel caso in cui tutti i nodi al livello superiore sono stati espansi; questo comporta che il cammino trovato sarà il più corto possibile in termine di numero di azioni, ma anche come costo se esso è unitario.

- Completezza: Arrivo alla soluzione meno profonda;
- Ottimalità: si, perché le azioni hanno tutte lo stesso costo;
- Complessità temporale: O(b^d);
- Complessità spaziale: O(b^{d-1}) nodi esplorati.

Questo tipo di ricerca è implementabile facilmente con una lista FIFO sulla frontiera.

La criticità di questo algoritmo è legata al fattore di ramificazione e alla elevata profondità, infatti, più passi sono richiesti per arrivare alla soluzione, più nodi devono essere esplorati e memorizzati nella lista aperta contenente la frontiera.

Ricerca Bidirezionale

La ricerca bidirezionale è un tipo di ricerca che ha come punti di partenza lo stato iniziale, ma anche quello finale. La prima ricerca parte dal nodo iniziale e procede dai suoi successori, mentre la seconda parte dal nodo finale, cercando di andare a ritroso verso il nodo di partenza. È come se si stessero eseguendo due ricerche in parallelo con opposti punti di fine e inizio.

La ricerca termina nel momento in cui le due "sotto-ricerche" si incontrano in un punto che sarà per entrambe parte della frontiera.

Rispetto a una ricerca in ampiezza, questo algoritmo gode di una minore complessità temporale.

- Completezza: arrivo alla soluzione meno profonda;
- Ottimalità: si, perché le azioni hanno tutte costo unitario;
- Complessità temporale: O($b^{d/2} + b^{d/2}$);
- Complessità spaziale: O($b^{d/2}$) nodi esplorati.

Nonostante la sua minore complessità, la ricerca bidirezionale ha lo stesso livello di completezza della ricerca in ampiezza, ma a differenza di questa richiede la conoscenza dello stato obiettivo che non sempre è dato.

Ricerca Informata AStar

La ricerca informata è una strategia di ricerca utilizzata per trovare una o più soluzioni a un problema che sfrutta a proprio vantaggio la conoscenza del problema trattato.

L'algoritmo è dotato quindi anche di una funzione di conoscenza (euristica) che permette di determinare l'ordine di ricerca in modo da ridurre il tempo o il costo di ricerca, stimando la probabilità di avvicinarsi/trovare la soluzione nei nodi selezionati e permettendo, di conseguenza, di scegliere il nodo con probabilità maggiore.

In particolare, la ricerca AStar, è una ricerca che si basa sia sul costo effettivo di cammino che sulla stima per raggiungere il nodo finale. La funzione di costo infatti è data da A(n) = F1(n) + F2(n), con F1 costo per raggiungere il nodo intermedio e F2 stima per il raggiungimento del nodo finale.

L'applicativo creato, in particolare, tratta due tipi di euristiche: la distanza di Hamming e la distanza di Manhattan.

Caratteristiche dell'algoritmo AStar:

- Completezza: Si, ma in spazi finiti;
- Ottimalità: se l'euristica è ammissibile su albero o consistente su grafo;
- Complessità temporale: O(b^{Δ});
- Complessità spaziale: O(b^{d}).

Astar con distanza di Hamming

Per quantificare matematicamente il concetto della separazione tra le parole di codice, si definisce la distanza di Hamming. La distanza di Hamming d (X, Y) tra due vettori X e Y di uguale lunghezza è il numero di valori differenti che compaiono nei due vettori nelle posizioni corrispondenti.

La distanza di Hamming applicata all'algoritmo AStar, quindi, si occupa di prendere come costo euristico il numero di celle diverse tra la soluzione e il puzzle, dopo aver applicato la mossa desiderata.

Sommando questa euristica al numero di mosse impiegate a raggiungere un determinato nodo, si ottiene la funzione di costo; l'algoritmo continuerà così a confrontare i costi ottenuti, scegliendo sempre di avanzare verso la mossa che comporterà un costo minore.

AStar con distanza di Manhattan

Dati due punti a una distanza d tra essi, la distanza di Manhattan, è la somma del valore assoluto delle differenze delle coordinate dei punti considerati; in particolare, in questo caso, la distanza di Manhattan è calcolata tra la posizione dell'elemento considerato e la sua corretta posizione contenuta nella soluzione.

Come avvenuto per la distanza di Hamming, la distanza di Manhattan viene applicata ogni volta che si vuole valutare la prossima mossa. Sommando questa distanza al numero di mosse impiegate a raggiungere la data posizione, si ricaverà la funzione di costo. Confrontando le funzioni di costo sulla frontiera, l'algoritmo andrà a scegliere il percorso con funzione di costo inferiore, fino a giungere alla soluzione.

IMPLEMENTAZIONE

Struttura base

Per poter eseguire tutti gli algoritmi, è stato necessario creare una classe che permettesse di simulare il comportamento del puzzle.

È stata quindi implementata la classe *structNode*.

Questa classe contiene come parametri la matrice che permette di tenere in considerazione la posizione di tutte le celle del puzzle, la stringa contenente tutte le mosse effettuate per arrivare in quella posizione e la posizione della cella vuota. Vengono poi implementate anche tutte le funzioni che permettono di effettuare i movimenti, prendendo in considerazione la cella vuota e tutte le celle vicine, la funzione clone che permette di clonare l'oggetto, la procedura di stampa dei movimenti e tutte le funzioni che permettono di restituire i vari dati inerenti al puzzle (posizione zero, mosse etc..).

Sono state inoltre implementate due classi per la visualizzazione grafica:

- userInterface: permette l'inserimento o la generazione casuale dei valori contenuti all'interno del puzzle, consente inoltre di scegliere la dimensione della tabella (tra 8 puzzle e 15 puzzle) e di eseguire i diversi algoritmi risolutivi;
- executeInterface: mostra a video i vari passaggi dell'esecuzione degli algoritmi e, nel caso si scelga di giocare, crea una semplice interfaccia che permette di cimentarsi nel gioco.

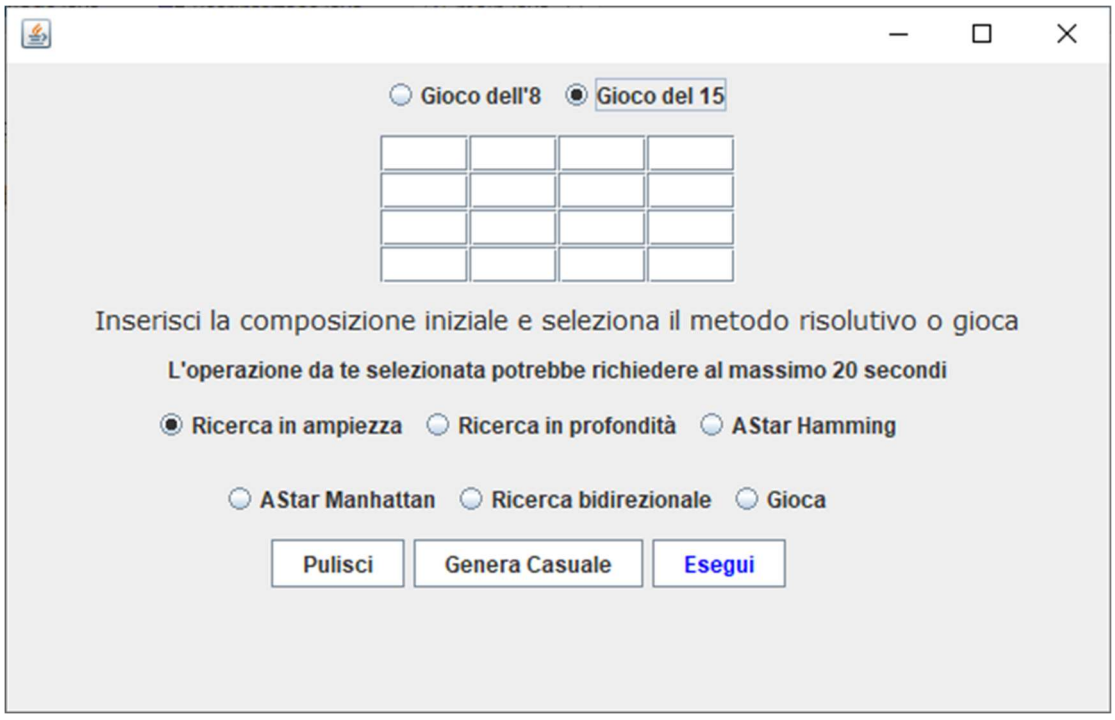

Figura 2- userInterface

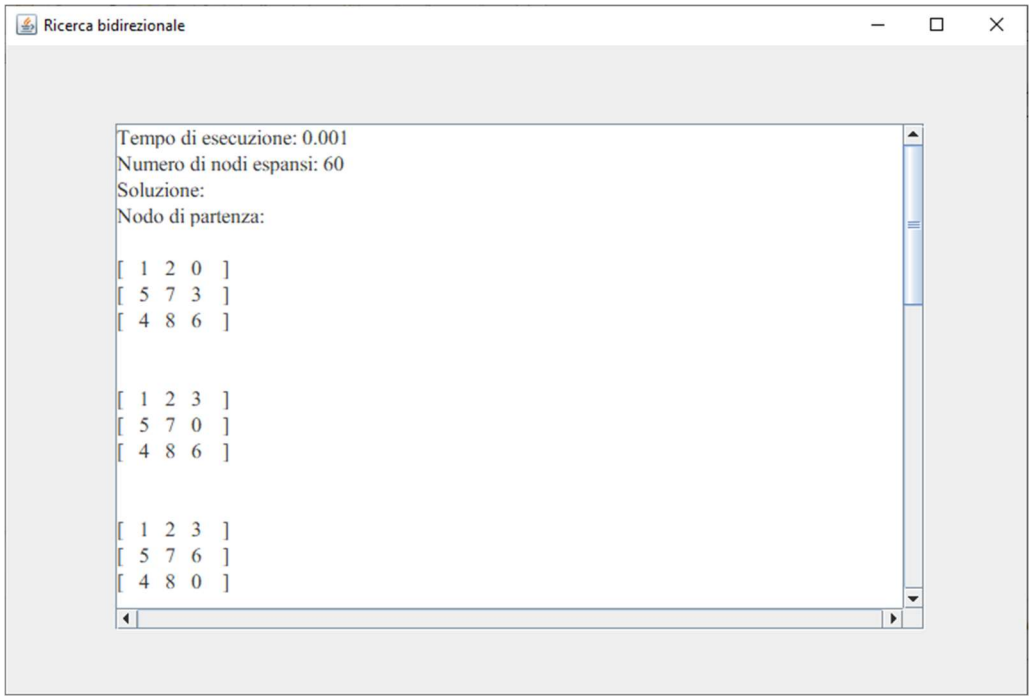

Figura 3- resultInterface

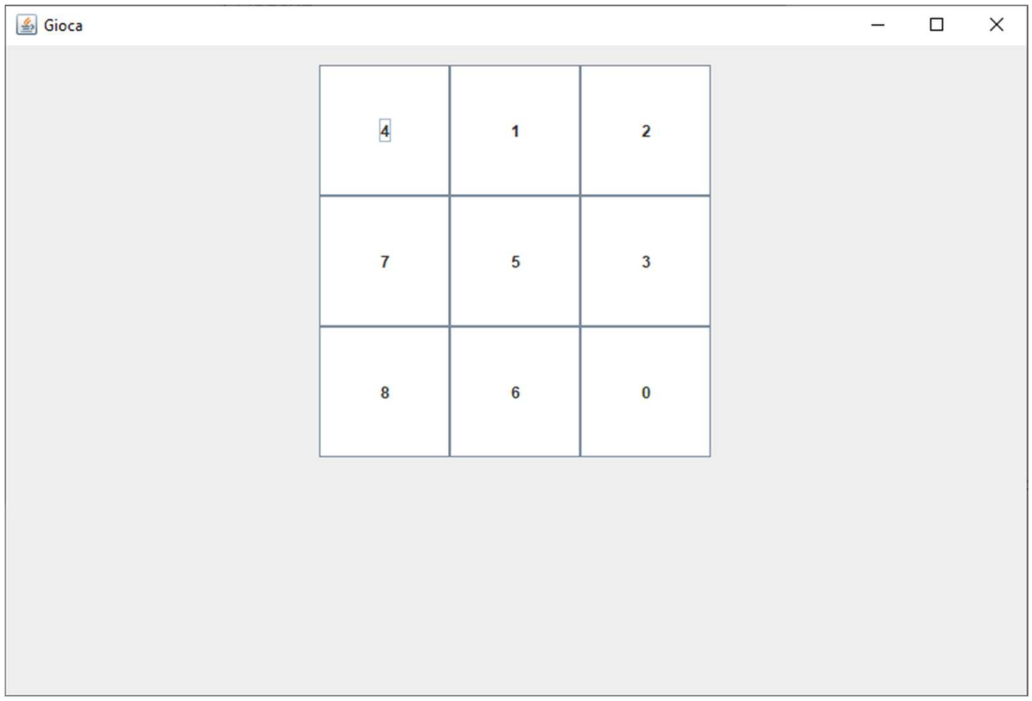

Figura 4- Gioca

Per semplicità e per permettere di comprendere il reale funzionamento del programma senza entrare nello specifico, di seguito, è mostrato lo pseudocodice.

A tutti gli algoritmi implementati è stato imposto un tempo massimo di esecuzione di 20 secondi, questo per evitare problemi di memoria oltre che per evitare tempi di attesa molto elevati in attesa della soluzione. Infatti, eseguendo gli algoritmi per più tempo, la memoria richiesta per la loro esecuzione diventava molto elevata. Questo potrebbe però svantaggiare le ricerche informate, visto che il calcolo delle euristiche richiede più tempo rispetto alla semplice esplorazione delle ricerche non informate.

Ricerca in Profondità

Per implementare questo algoritmo esistono diverse possibilità, di seguito sono riportate due delle principali: la variante con ricorsione e quella senza.

Pseudocodice con ricorsione:

```
// Depth First Search Algorithm
1  funzione DFSRecursive(G,v):
2      label v as discovered
3      for all edges from v to w in G.adjacentEdges(v) do
4          if vertex w is not labeled as discovered then
5              recursively call DFSRecursive(G,w)
```

L'algoritmo effettivamente utilizzato non comprende però la ricorsione. È stata effettuata questa scelta per mantenere un'implementazione simile in tutti i metodi.

Pseudocodice senza ricorsione:

```
// Depth First Search Algorithm
1  funzione DFS(v):
2      label v as discovered
3      while not findSolution do
4          for all child nodes do
5              if vertex w is not labeled as discovered then
6                      add vertex at border
7          extract node from border
8          if node is solution then
9              findSolution=True
```

Ricerca in Ampiezza

Pseudocodice

```
// Breadth First Search Algorithm
1  funzione BFS(G,v):
2      create tail Q
3      insert v in Q
4      mark v
5      while Q not Empty:
6          t ← Q.removeFromTail()
7          if t is solution then:
8              return t
9          for all children do
12             u ← G.adjacentNodes(t)
13             if u not border o u not analyzed then:
14                  mark u
15                  insert u in Q
16     return none
```

Ricerca Bidirezionale

Pseudocodice

```
// Bidirectional Algorithm

1    funzione BS(v,end):
2      create tail Q
3      create tail Q1
4      insert v in Q
5      insert end in Q1
6      mark v
7      while Q not empty:
8          t ← Q.removefromtail()
9          if t contained in Q1 then:
10              mergePath()
11             return solution
12          for all children do
13             u ← G.adjacentNodes(t)
14             if u not border then:
15                  mark u
16                  insert u in Q
17              t ← Q1.removefromtail()
18          if t contain Q then:
19              mergePath()
20                      return solution
21          for all children do
22             u ← G.adjacentNodes(t)
23             if u not contained in Q1 or u not marked then:
24                  mark u
25                  insert u in Q
26     return none
```

Ricerca AStar

Lo schema base della ricerca AStar è mostrato di seguito. La parte che differisce nelle due implementazioni è semplicemente l'esecuzione dell'euristica.

Pseudocodice ricerca AStar:

```
// A* Search Algorithm
1  Initialize the openlist
2  Initialize the closedlist
3    put the starting node in openlist

3  while openlist not empty
4    Remove element from openlist
5          for all possibleMoves do
6             F2(n)=calculateEuristicFunction()
7              F1(n)calculateCost()
8              A(n)=F1(n)+F2(n)
9              If solution
10                Return solution
11          selectLessExpensive(A)
12     push element in closedlist
13    end
```

Calcolo distanza di Hamming

Pseudocodice del calcolo dell'euristica usando la distanza di Hamming:

```
// Hamming distance
1  Inizialize cost=0
2    for all matrix elements do
3              If element != solution then
4                 cost++
5     Return cost;
6    end
```

Calcolo distanza di Manhattan

Pseudocodice del calcolo dell'euristica usando la distanza di Manhattan:

```
// Manhattan distance
1  Inizialize cost=0
2    for all matrix elements do
3              If element != solution then
4                 cost+= abs(element pos-solution pos)
5     Return cost;
6    end
```

CONFRONTO E RACCOLTA DATI

Per analizzare i 5 algoritmi sono state generate 30 prove in entrambe le versioni del gioco di modo da avere un campione consistente su cui effettuare le valutazioni. Di seguito sono riportati i risultati di queste prove, in particolare il numero di cicli richiesti per giungere alla soluzione del problema.

Tabella 1- Risultati gioco dell'8

gioco 8							
passi							
prof	altri	Caso	Profondità	Ampiezza	Bidirezionale	AStar Hamming	AStar Manhattan
na	16	1	na	8451	141	5139	5139
na	9	2	na	275	17	271	271
na	15	3	na	5867	119	4232	4232
na	5	4	na	30	7	30	30
na	16	5	na	8450	153	5137	5137
na	2	6	na	3	1	3	3
60	8	7	60	153	15	147	147
na	8	8	na	98	15	97	97
na	8	9	na	149	14	139	139
430	4	10	451	15	3	15	15
35	7	11	34	48	7	48	48
na	9	12	na	149	14	139	139
34	8	13	36	78	13	68	68
29429	13	14	31496	1360	51	1013	1013
5	5	15	4	8	3	8	8
27	4	16	27	8	2	8	8
31692	14	17	33973	4164	85	2772	2772
3	3	18	3	5	2	5	5
6	6	19	6	32	6	32	32
na	10	20	na	459	30	403	403
na	7	21	na	71	11	81	81
na	10	22	na	456	30	400	400
na	2	23	na	3	1	3	3
26	4	24	26	12	3	12	12
823	9	25	869	262	24	260	260
na	12	26	na	1455	51	1094	1094
na	9	27	na	275	17	271	271
430	4	28	451	15	3	15	15
21479	15	29	22982	7258	158	4464	4464
3	3	30	3	5	2	6	6
		Media	6028,066667	1320,4667	33,26666667	877,0666667	877,0666667
		Prove fallite	15	0	0	0	0
		% successo	50	100	100	100	100
		cicli totali	90421	39614	998	26312	26312

Come si può notare dai risultati riportati nella *Tabella 1,* l'algoritmo che ha dato i migliori risultati in termini di cicli è stata la ricerca bidirezionale. Anche tenendo in considerazione il fatto di dover espandere la frontiera in entrambi i sensi (a partire dalla soluzione e dalla configurazione di partenza), in questo particolare problema, è la soluzione migliore.

AStar ha prestazioni migliori della ricerca in ampiezza poiché esegue un quantitativo di cicli e di esplorazioni inferiore, richiedendo quindi l'utilizzo di una quantità minore di memoria. Nei casi in cui la configurazione di partenza è molto lontana dalla soluzione del problema, questi vantaggi sono maggiormente visibili (casi 1, 3, 5, 17, 26 e 29).

I due tipi di ricerca AStar implementati, nonostante le differenti euristiche usate, restituiscono sempre lo stesso numero di mosse per arrivare alla soluzione. Questo è probabilmente dovuto al fatto che le mosse possibili sono molto limitate (nel migliore dei casi, quattro possibili spostamenti). L'unica differenza notata tra le due euristiche è relativa al tempo di esecuzione, che in casi sporadici è minore di qualche millisecondo nell'AStar avente come euristica la distanza di Hamming, probabilmente per via della minor complessità di calcolo della stessa.

Nonostante siano state prese delle precauzioni per evitare l'ingresso in loop, introducendo una lista contenente gli stati visitati, la ricerca in profondità ha una percentuale di successo del 50%, inoltre le soluzioni trovate sono raramente ottimali. Molto spesso, infatti, il numero di mosse necessarie per giungere alla soluzione adoperando questo tipo di ricerca, è molto più alto rispetto agli altri algoritmi che invece trovano sempre la soluzione ottimale, senza contare che il numero medio di nodi da esplorare è decisamente più alto rispetto a qualsiasi altro.

Tabella 2- Risultati gioco del 15

gioco 15							
Numero passi soluzione							
Profondità	Altri		Profondità	Ampiezza	Bidirezionale	AStar Hamming	AStar Manhattan
1821	9	1	1884	447	18	440	440
60	12	2	60	4136	43	3763	3763
na	6	3	na	86	7	86	86
na	8	4	na	263	11	247	247
na	10	5	na	1328	28	1235	1235
na	17	6	na	346479	805	na	na
na	18	7	na	na	447	na	na
na	11	8	na	4019	69	3655	3655
na	9	9	na	862	40	834	834
na	13	10	na	18186	196	15110	15110
na	12	11	na	8583	94	7511	7511
na	12	12	na	10358	111	8896	8896
na	9	13	na	817	38	1048	1048
na	9	14	na	666	18	650	650
na	na	15	na	na	na	na	na
3	3	16	3	4	2	4	4
1821	9	17	1884	447	18	440	440
na	17	18	na	na	657	na	na
na	8	19	na	395	18	511	511
na	6	20	na	69	7	69	69
3	3	21	3	4	2	4	4
na	15	22	na	99954	318	na	na
na	10	23	na	2275	52	2123	2123
na	na	24	na	na	na	na	na
na	23	25	na	na	4346	na	na
na	6	26	na	81	6	82	82
na	12	27	na	8583	94	7511	7511
na	21	28	na	na	3110	na	na
na	23	29	na	na	6959	na	na
na	4	30	na	12	3	16	16
		Media	766,8	22089,304	625,6071429	2582,619048	2582,619048
		Prove fallite	25	7	2	9	9
		% successo	16,67	76,67	93,333	70	70
		cicli totali	3834	508054	17517	54235	54235

La *Tabella 3* conferma sostanzialmente i risultati della *Tabella 1.*

Il miglior algoritmo si conferma la ricerca bidirezionale che ha successo nel 93% dei casi, con una media di soli 625 nodi esplorati.

Gli algoritmi AStar si confermano al secondo posto in quanto a efficienza, esplorando mediamente solo il 10% dei nodi dalla ricerca in ampiezza, la quale però ha un vantaggio in velocità dovuto al mancato calcolo delle euristiche, che, nonostante forniscano un vantaggio in termini di efficienza, comportano una velocità di esplorazione inferiore.

La ricerca in ampiezza garantisce dei buoni risultati nonostante la gran quantità di cicli e memoria necessaria. In problemi più complessi però, diventerebbe sostanzialmente impossibile da utilizzare.

La ricerca in profondità ha ottenuto i risultati peggiori con appena il 16% di casi di successo, diventando praticamente inutilizzabile per risolvere questo genere di problema, senza contare che le soluzioni ricavate attraverso l'uso di questo metodo, non sono ottime come è ben possibile vedere nei casi 1, 2 e 17.

CONCLUSIONI

Dall'analisi dei dati e delle prestazioni si può osservare che, nel breve periodo e per quanto riguarda questo particolare ambito applicativo, la ricerca bidirezionale si è rivelata la miglior scelta in termini di efficienza e velocità.

Il numero di possibili mosse ridotto e l'ambiente relativamente piccolo ha penalizzato le ricerche informate, che hanno comunque ottenuto delle buone prestazioni. In altri ambienti più complessi avrebbero sicuramente surclassato le altre tipologie di ricerca, sia in termini di velocità che di efficienza.

Personalmente ho trovato molto interessante questo progetto perché mi ha permesso di vedere, dal punto di vista pratico e dell'implementazione attraverso il codice, questi algoritmi.

BIBLIOGRAFIA

Francesco Trovò, Slide corso intelligenza artificiale Università degli studi di Bergamo

Stuart Russell - Peter Norvig, Intelligenza artificiale: Un approccio moderno

Okpedia: enciclopedia online