FONDAMENTI DI INTELLIGENZA ARTIFICIALE

AI-Puzzle

Raffaele Sansone

https://github.com/RaffaeleSansone/AI-Puzzle

Indice

Capitolo 1

Introduzione

1.1 Cos'è Ai-Puzzle ?

Il progetto Ai-Puzzle ha come obiettivo la creazione di una semplice intelligenza artificiale capace di risolvere il gioco Puzzle 8.

Questo Puzzle game consiste nella risoluzione di un rompicapo numerico *(figura 1.1)* composto da 8 tessere che devono essere posizionate in ordine crescente per vincere la partita. Il tavolo da gioco è composto da una griglia 3x3 in cui manca una tessera. Lo spazio vuoto che si viene a creare ha il compito di far scorrere le altre caselle per formare la composizione corretta.

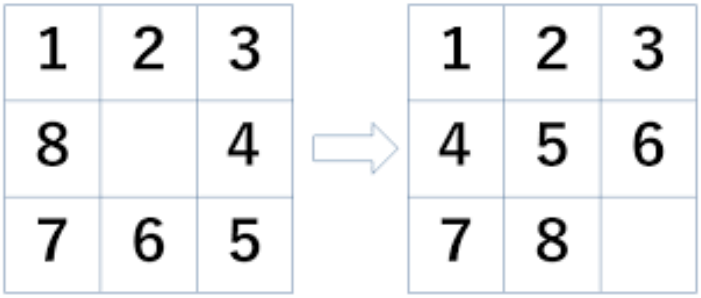

Figura 1.1: Puzzle 8

1.2 Scopo del Progetto

Lo scopo di questo progetto è la creazione di un agente intelligente in grado di risolvere il puzzle game tramite l'utilizzo di algoritmi di ricerca.

Si vuole mettere in evidenza come l'uso di diversi algoritmi può portare a risultati più o meno ottimali per la risoluzione dello stesso problema.
In particolare sono stati scelti gli algoritmi di:

- Ricerca in Profondità
- Ricerca Bidirezionale
- Ricerca A*

Si è scelto, inoltre, di implementare una demo giocabile utilizzando il linguaggio *JAVA*. La demo avrà il compito di mettere in luce le differenze prestazionali dei 3 algoritmi utilizzati.

1.3 Progettazione

La progettazione è partita dall'individuazione delle *specifiche PEAS*, ovvero tutte le specifiche che compongono l'ambiente operativo dell'agente e dalle proprietà dell'ambiente (*environment*).

- **Performance:** La capacità dell'agente di risolvere il puzzle
- **Ambiente:** Il tavolo da gioco dove sono disposte le tessere
- **Attuatori:** La funzione che permette lo spostamento delle tessere per arrivare alla soluzione
- **Sensori:** Interfaccia che da l'input all'agente. Nello specifico la combinazione di tessere mischiate da ordinare

Basandoci sulla descrizione del gioco è possibile esplicitare le caratteristiche dell'ambiente.

- **Completamente Osservabile:** Lo stato dell'ambiente è noto all'agente in qualsiasi momento.

- **Deterministico:** Lo stato successivo è completamente calcolabile dallo stato attuale e dalla specifica azione scelta.
- **Sequenziale:** La sequenza di mosse fatte in precedenza è rilevante per lo stato attuale.
- **Statico:** L'ambiente rimane invariato se non è l'agente ad effettuare azioni.
- **Discreto:** L'ambiente fornisce un numero limitato di percezioni e azioni distinte.
- **Singolo:** L'ambiente consente la presenza di un unico agente.

Si è passati, quindi, alla progettazione dell'agente (*figura 1.2*)

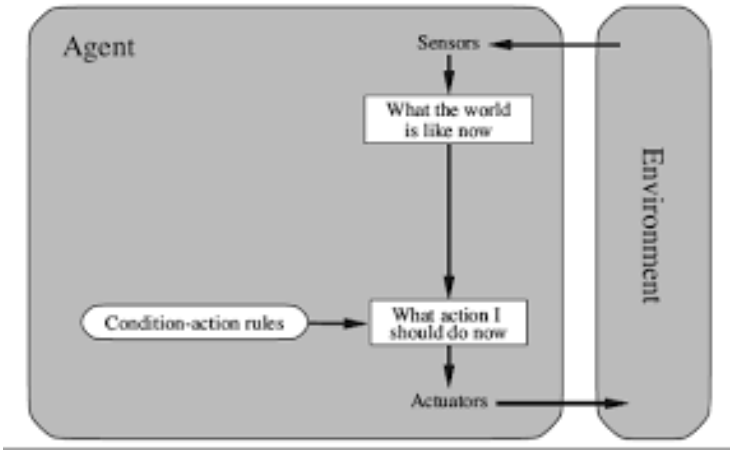

Figura 1.2: Progettazione ad alto livello dell'agente

L'ambiente è caratterizzato dalla presenza dell'utente, infatti è l'utente che passa all'agente la sequenza da cui far iniziare la risoluzione del puzzle. L'agente preleva questa richiesta tramite i sensori, calcola il risultato e tramite gli attuatori notifica all'utente la combinazione vincente. Sensori e attuatori sono concetti astratti e poiché stiamo realizzando un software verranno implementati come parti di questo.
L'agente è composto da una parte funzionale e da un'interfaccia grafica basilare. Entrambe le parti sono state sviluppate utilizzando *JAVA* e *Eclipse IDE*.

Capitolo 2

Realizzazione

2.1 Stato del nodo

Come prima cosa si è scelto in che modo rappresentare lo stato; è stato utilizzato un array unidimensionale contenente nove valori. Per convenzione si è deciso di rappresentare la casella vuota con lo zero.

[1, 2, 3, 8, 0, 4, 7, 6, 5]

(rappresentazione stato iniziale figura 1.1)

A questo punto possiamo definire lo stato finale, ovvero la configurazione del tavolo da gioco che per convenzione consideriamo vincente:

[1, 2, 3, 4, 5, 6, 7, 8, 0]

(rappresentazione stato finale figura 1.1)

2.2 Spostamento

Per la generazione di nuovi stati deve essere cambiata la configurazione del tavolo da gioco. Gli spostamenti vengono effettuati a partire dalla cella zero. Quest'ultima potrà spostarsi per un massimo di quattro direzioni *(su, destra, giù, sinistra)*.

Per fare ciò è stato utilizzato il metodo *generateNextNode()*. Questo genera nuove configurazioni del tavolo da gioco tenendo presente la posizione della cella contenente lo zero. In base alla sua posizione la cella zero ha limiti nello spostamento. Nello specifico:

- La cella zero può effettuare 4 spostamenti se è al centro del tavolo da gioco.
- La cella zero può effettuare 2 spostamenti se è in un angolo.
- La cella zero può effettuare 3 spostamenti se è al centro di una riga (che non sia il centro del tavolo da gioco).

generateNextNode() a sua volta si serve di un altro metodo per conoscere la posizione della cella zero (*getEmptyDowel()*), dopodichè chiamando *positionChange()* a cui vengono passati:

- La nuova posizione della cella zero
- La posizione attuale della cella zero
- Il nuovo nodo

effettua uno spostamento valido.

```java
public ArrayList<StateNode> generateNextNode() {

    ArrayList<StateNode> nextNode = new ArrayList<StateNode>();
    int emptyDowel = getEmptyDowel();

    // Proviamo a generare un nuovo stato facendo scorrere il tassello '0' sulla sinistra
    if (emptyDowel != 0 && emptyDowel != 3 && emptyDowel != 6) {
        /*
         * Se siamo qui significa che è possibile andare a sinistra.
         * Viene generato un nuovo stato
         */
        positionChange(emptyDowel - 1, emptyDowel, nextNode);
    }

    // Proviamo a generare un nuovo stato facendo scorrere il tassello '0' verso il basso
    if (emptyDowel != 6 && emptyDowel != 7 && emptyDowel != 8) {
        /*
         * Se siamo qui significa che è possibile andare in basso.
         * Viene generato un nuovo stato
         */
        positionChange(emptyDowel + 3, emptyDowel, nextNode);
    }

    // Proviamo a generare un nuovo stato facendo scorrere il tassello '0' verso l'alto
    if (emptyDowel != 0 && emptyDowel != 1 && emptyDowel != 2) {
        /*
         * Se siamo qui significa che è possibile andare in alto.
         * Viene generato un nuovo stato
         */
        positionChange(emptyDowel - 3, emptyDowel, nextNode);
    }
```

Figura 2.1: Estratto della classe StatePuzzleGame

2.3 La classe SearchNode

Questa classe rappresenta ogni nodo visitato durante il percorso per arrivare alla soluzione finale. Essa è stata fornita di due metodi costruttori: uno per il nodo radice e l'altro per tutti i nodi sottostanti. I metodi di questa classe sono essenziali per salvare tutte le informazioni rilevanti su ogni nodo.

- **getCurrentState()** restituisce la stato corrente
- **getParentState()** restituisce se presente lo stato genitore del nodo
- **getCost()** restituisce il costo dello stato corrente
- **getFCost()** restituisce il costo effettivo per arrivare al nodo n sommato al costo dell'euristica per arrivare al nodo finale.
- **getHCost()** restituisce il costo dell'euristica

```java
public class SearchNode {
    /**
     * Costruttore per il nodo radice
     *
     * @param state è lo stato che viene passato
     */
    public SearchNode(StateNode state) {
        currentState = state;
        parentState = null;
        cost = 0;
        hCost = 0;
        fCost = 0;
    }

    /**
     * Costruttore per tutti gli altri nodi
     *
     * @param prevState é il nodo genitore
     * @param state     è lo stato del nodo
     * @param c         il costo g(n) per raggiungere il nodo
     * @param h         il costo h(n) per raggiungere il nodo
     */
    public SearchNode(SearchNode prevState, StateNode state, double c, double h) {
        parentState = prevState;
        currentState = state;
        cost = c;
        hCost = h;
        fCost = cost + hCost;
    }
```

Figura 2.2: Costruttori della classe SearchNode

2.4 Ricerca in Profondità

La ricerca in profondità (*figura 2.3*) è un tipo di ricerca non informata, cioè una ricerca che non utilizza alcun tipo di conoscenza del problema al di là della definizione del problema stesso. Essa espande i nodi a partire dai più profondi dell'albero di ricerca e vengono analizzati uno alla volta.

Figura 2.3: Ricerca in profondità

Nella ricerca in profondità ogni singolo ramo viene espanso completamente prima di analizzare il successivo. Una volta analizzata la ramificazione di un ramo, la memoria può essere liberata per fare spazio all'analisi del ramo successivo.

Questo algoritmo risulta *completo e ottimo*, in quanto riesce sempre a trovare una soluzione, se esiste e, garantisce di trovare anche la soluzione ottimale.

2.4.1 Esecuzione ricerca in profondità

L'esecuzione dell'algoritmo schematicamente compie i seguenti passaggi:

1. Vengono inizializzati il nodo radice e il nodo obiettivo
2. Viene effettuato un controllo per verificare se il nodo corrente è il nodo obiettivo. Se lo è viene restituita la soluzione.
3. Se non è un nodo obiettivo vengono espansi i nodi successori

4. Viene controllato se i nodi sono stati già ispezionati, solo se superano il controllo vengono aggiunti nella coda.

5. Si riparte dal punto 2 finchè non si trova la soluzione.

2.4.2 Considerazioni sulla ricerca in profondità

Applicando questo tipo di algoritmo al problema in questione risulta evidente come al crescere della quantità di nodi da esplorare diminuisca l'efficienza dell'algoritmo in termini di *complessità temporale* e *complessità spaziale*.

Dato un fattore di ramificazione **b** e una fattore di profondità massima **d**, la quantità di memoria necessaria per l'algoritmo è pari a **b d**. La durata d'esecuzione, invece, è stimata $\boldsymbol{O(b^d)}$.

E' da considerare anche la possibilità di imbattersi in *cammini ridondanti* che andrebbero ad inficiare ulteriormente sulle prestazioni. Per evitare questo è stato aggiunto un metodo *checkNodes()* che verifica se un nodo è già stato visitato (*figura 2.4*).

```
/*
 * Metodo che verifica se un nodo è già stato valutato
 */
private static boolean checkNodes(SearchNode n)
{
    boolean isEqual = false;
    SearchNode checkNode = n;

    // Verifica se lo stato genitore è uguale allo stato corrente
    while (n.getParentState() != null && !isEqual)
    {
        if (n.getParentState().getCurrentState().equals(checkNode.getCurrentState()))
        {
            isEqual = true;
        }
        n = n.getParentState();
    }

    return isEqual;
}
```

Figura 2.4: Metodo checkNodes()

Tirando le somme, anche con l'aggiunta di *checkNodes()*, la ricerca in profondità risulta essere abbastanza inefficiente per la risoluzione del puzzle game quando lo spazio degli stati è di dimensioni elevate.

Questo si evince dall'utilizzo della demo che è stata provata con le seguenti combinazioni:

Input 1: [1, 2, 3, 4, 5, 6, 0, 7, 8], **Input 2:** [0, 1, 2, 3, 4, 5, 6, 7, 8]

Input 3: [1, 3, 4, 0, 2, 5, 6, 7, 8], **Input 4:** [1, 2, 0, 7, 8, 6, 5, 3, 4]

Input 5: [1, 2, 3, 4, 5, 0, 7, 8, 6]

Dall' esecuzione con questi input sono stati generati i seguenti risultati:

Input	Costo	Iterazioni	Tempo in millisecondi
1	2.0	2	$<$ 0ms
2	1843086.0	70364	113992ms
3	_*	_*	_*
4	_*	_*	_*
5	576.0	41	5ms

(* = nessuna soluzione in un tempo accettabile)

2.5 Ricerca Bidirezionale

La ricerca bidirezionale parte da due nodi, un nodo iniziale e un nodo finale (*nodo obiettivo*), l'algoritmo effettua due ricerche parallele. La prima ricerca scandaglia i cammini a partire dal nodo iniziale sui nodi successori; la seconda ricerca scandaglia a ritroso i cammini a partire dal nodo finale sui nodi predecessori; l'algoritmo, infine, verifica l'esistenza di punti di contatto a metà strada tra le due ricerche.

Per trovare il percorso tra il nodo di partenza e il nodo di destinazione, l'algoritmo bidirezionale verifica la coincidenza dei nodi sulla frontiera delle due ricerche. Ad esempio, il nodo A e il nodo B compaiono sulla frontiera di entrambe le ricerche. Una volta individuati i nodi comuni sulla frontiera, si può facilmente individuare il cammino completo dal nodo iniziale al nodo finale.

Pur avendo una minore complessità, l'algoritmo di ricerca bidirezionale non penalizza *la completezza* della ricerca. Le due ricerche parallele consentono

di raggiungere il medesimo livello di completezza dell'algoritmo precedente in un tempo di elaborazione inferiore.

2.5.1 Esecuzione ricerca bidirezionale

L'esecuzione dell'algoritmo si serve della *ricerca in ampiezza* per arrivare alla soluzione finale. La ricerca in ampiezza è una ricerca non informata in cui i nodi sono analizzati seguendo un ordine di vicinanza al nodo radice. Sono espansi dapprima i nodi più vicini alla radici, successivamente tutti gli altri nodi successori fino ad una profondità **d**.

L'esecuzione dell'algoritmo compie schematicamente i seguenti passaggi:

1. Vengono inizializzate due code, una con lo stato iniziale e una con lo stato obiettivo.
2. Si fanno partire i due algoritmi di ricerca in ampiezza fino a che non si trova uno stato comune alle due code
3. una volta trovato si stampano i due percorsi che hanno portato ad esso, altrimenti si continua con l'esecuzione
4. L'esecuzione va avanti fino a che non viene trovato lo stato in comune.

2.5.2 Considerazioni sulla ricerca bidirezionale

Applicando questo algoritmo al puzzle game, risulta evidente il miglioramento in termini di *complessità temporale* rispetto alla ricerca in profondità.

Dato un fattore di ramificazione **b** e una fattore di profondità massima **d**, l'algoritmo dimezza i tempi di esecuzione, in quanto partono due ricerche distinte che si interrompono a metà strada. La complessità risulta quindi $\boldsymbol{O(b^{d/2})}$ per ogni ricerca.

Anche in questo caso, per evitare cammini ridondanti, è stato utilizzato il metodo *checkNodes()*, per verificare se un nodo è già stato valutato in precedenza.

Tirando le somme, questo tipo di ricerca risulta nettamente migliore per il problema del puzzle game rispetto all'algoritmo precedente. Questo si evince dall'utilizzo della demo che ha generato i seguenti risultati:

Input 1: [1, 2, 3, 4, 5, 6, 0, 7, 8], **Input 2:** [0, 1, 2, 3, 4, 5, 6, 7, 8]

Input 3: [1, 3, 4, 0, 2, 5, 6, 7, 8], **Input 4:** [1, 2, 0, 7, 8, 6, 5, 3, 4]

Input 5: [1, 2, 3, 4, 5, 0, 7, 8, 6]

Input	Costo	Iterazioni	Tempo in millisecondi
1	2.0	3	2ms
2	180.0	2248	47ms
3	94.0	793	17ms
4	_*	_*	_*
5	0.0	1	1ms

(* = nessuna soluzione in un tempo accettabile)

2.6 Ricerca A*

La ricerca A* è una strategia di ricerca informata in cui la scelta del cammino si basa sia sul costo effettivo **g(n)**, necessario per raggiungere un nodo intermedio n dal nodo di origine, sia sulla stima del costo **h(n)** necessario per raggiungere il nodo finale (*obiettivo*) a partire dal nodo intermedio n. La funzione di ricerca è, quindi, composta da due funzioni di costo:

$$\mathbf{F(n) = g(n) + h(n)}$$

La ricerca A* analizza il costo del cammino **g(n)** soltanto fino ad un livello di profondità n (*nodo intermedio*) e da qui stima il cammino **h(n)** per raggiungere il nodo obiettivo. In tal modo la ricerca A* permette di analizzare dapprima i cammini presumibilmente migliori, ossia quelli in grado di minimizzare la somma delle funzioni di costo **g(n)** e **h(n)**.

Questo algoritmo risulta essere *ottimale* quando **h(n)** (*euristica*) è ammissibile e consistente. L'euristica assume un ruolo chiave nelle prestazioni dell'algoritmo, infatti deve essere scelta con cura per ottenere i cammini migliori.

Altre caratteristiche della ricerca A* sono: la *completezza*, se il numero dei nodi risulta finito e la *complessità temporale* che risulta essere bassa a favore di una buona euristica.

2.6.1 Esecuzione ricerca A*

L'esecuzione dell'algoritmo è basata sulla scelta dell'euristica **h(n)**. Per minimizzare i tempi di ricerca è stato utilizzato un metodo *setOutPlace()* (*figura 2.5*) che tiene conto delle caselle fuori posto rispetto allo stato obiettivo. In questo modo l'algoritmo sceglie sempre di seguire il percorso con un minor numero di caselle fuori posizione.

```
/*
 * Metodo per settare i tasselli fuori posto
 */
private void setOutPlace() {
    for (int i = 0; i < currentBoardState.length; i++) {
        if (currentBoardState[i] != WIN_STATE[i]) {
            outPlace++;
        }
    }
}
```

Figura 2.5: Metodo per tenere traccia delle caselle fuori posto

L'esecuzione dell'algoritmo compie schematicamente i seguenti passaggi:

1. Viene inizializzato il nodo radice e aggiunto a una coda.
2. Viene controllato se è un nodo obiettivo, se non lo è vengono generati i nodi successori
3. A questo punto si tiene conto del costo dell'euristica e solo i nodi con il minor numero di caselle fuori posto vengono inseriti nella coda per essere valutati
4. Si riparte dal punto due fino a che non si arriva al nodo obiettivo.

2.6.2 Considerazioni sulla ricerca A*

Applicando questo algoritmo al puzzle game, risulta evidente quanto l'euristica giochi un ruolo chiave per ridurre la *complessità temporale* dell'algoritmo.

Anche in questo caso, per evitare cammini ridondanti, è stato utilizzato il metodo *checkNodes()* per verificare se un nodo è già stato valutato in precedenza.

Tirando le somme, questo tipo di ricerca potrebbe risultare migliore per risolvere il puzzle game rispetto agli algoritmi usati in precedenza. In questo caso l'euristica ci permette di scartare molti nodi che negli algoritmi precedenti venivano visitati, tuttavia bisogna valutare se la scelta dell'euristica ci porterà ad una soluzione con un minor utilizzo di risorse.

L'utilizzo della demo che ha generato i seguenti risultati:

Input 1: [1, 2, 3, 4, 5, 6, 0, 7, 8], **Input 2:** [0, 1, 2, 3, 4, 5, 6, 7, 8]

Input 3: [1, 3, 4, 0, 2, 5, 6, 7, 8], **Input 4:** [1, 2, 0, 7, 8, 6, 5, 3, 4]

Input 5: [1, 2, 3, 4, 5, 0, 7, 8, 6]

Input	Costo	Iterazioni	Tempo in millisecondi
1	2.0	2	3ms
2	502.0	6625	63ms
3	262.0	672	392ms
4	_*	_*	_*
5	0.0	1	3ms

(* = nessuna soluzione in un tempo accettabile)

Capitolo 3

Conclusioni

3.1 Analisi dei diversi algoritmi

Tra gli algoritmi esaminati, quello con maggiore efficienza è risultato *l'algoritmo di ricerca bidirezionale*. Questo algoritmo grazie alla doppia ricerca in ampiezza, è riuscito a dimezzare i tempi di esecuzione, riuscendo ad ottenere quasi sempre buoni risultati.

La ricerca A*, invece, ha deluso le aspettative a causa di un uso non ottimale dell'euristica scelta per l'algoritmo. La selezione degli stati con un minor numero di caselle fuori posto non ha portato ad una più rapida soluzione del problema, infatti, come si evince dai test effettuati, la ricerca A* ha avuto un peggior andamento rispetto alla ricerca bidirezionale.

La ricerca in profondità è stata quella ad ottenere più difficoltà nella risoluzione del problema. L'esplorazione di tutti i nodi dell'albero non ha prodotto in più di un caso un risultato.

Per tutti gli algoritmi presi in esame è risultato come ad alcuni input iniziali non si sia trovata una soluzione in tempi accettabili perchè lo spazio degli stati era troppo esteso (Nel caso della ricerca A* con l'input 4 si è avuto un *OutOfMemory*).

Con uno spazio degli stati di grandi dimensioni la memoria occupata inizia a essere un problema e sarebbe opportuno usare tecniche di *memory-bounded heuristic search* come IDA* o SMA*.