# Algoritmo A* per la risoluzione del 15-puzzle

Eleonora Ristori

Maggio 2022

## Indice



## 1 Introduzione

Il progetto consiste in un'analisi delle prestazioni dell'algoritmo di ricerca informata A* per la risoluzione del 15-puzzle. Il puzzle, infatti, prevede circa $10^{13}$ diverse combinazioni (16 !/2 visto che la metà delle combinazioni non è raggiungibile da uno stato dato, facendo scorrere le tessere sulla griglia) e ciò rende impossibile una ricerca sistematica per la determinazione della soluzione. Si rende quindi necessario un algoritmo informato come A* che, sfruttando euristiche più o meno efficienti, riesce a determinare la soluzione in un tempo abbastanza ridotto. Per la realizzazione del programma è stato usato come guida il codice python in https://github.com/aimacode

# 2 L'algoritmo A*

## 2.1 A* monodirezionale

A* è un algoritmo di ricerca informata che sfrutta diverse euristiche con l'obiettivo di giungere più rapidamente alla soluzione rispetto ad un normale algoritmo di ricerca. In questa analisi è stato implementato come caso particolare dell'algoritmo di Best First Search in cui la funzione che determina l'elemento "migliore" è data da:

$$f(n) = path\_cost + h(n)$$

Dove il path cost rappresenta il numero di mosse necessarie per raggiungere lo stato considerato da quello iniziale e h(n) è un'euristica.

## 2.2 A* bidirezionale

L'algoritmo A* bidirezionale prevede che la ricerca simultaneamente proceda sia direttamente dallo stato iniziale verso il goal sia viceversa dal goal verso lo stato iniziale. Questo si traduce in un guadagno se i due alberi di ricerca si incontrano poichè, dato b il branching factor e d la profondità raggiunta, il costo $b^{d/2} + b^{d/2}$ è molto minore di $b^d$. L'algoritmo prevede la creazione di un nuovo problema con gli stati iniziale e finale invertiti. Sono quindi previste due frontiere e due tabelle in cui si memorizzano i nodi raggiunti da ciascuna ricerca. Ad ogni iterazione viene scelta la frontiera da espandere sulla base dell'euristica fornita. Viene quindi eseguita la funzione proceed() che, in primo luogo, controlla se le due ricerche si siano incontrate, accedendo alla lista contenente i nodi raggiunti dalla ricerca parallela. Se ciò non si è verificato, il nodo viene espanso ed i figli vengono aggiunti alla frontiera se è la prima volta che sono stati raggiunti (e in tal caso anche all'insieme dei nodi raggiunti) o se il costo stimato dall'euristica risulta minore di quello calcolato in precedenza per quello stato raggiunto attraverso un diverso percorso. La funzione adaptPath() serve infine a ricostruire il cammino dalla radice fino al goal. La funzione semplicemente ricava il percorso invertendo quello della ricerca diretta ed aggiungendo in coda quello della ricerca inversa.

## 2.3 La terminazione della ricerca

Con l'algoritmo bidirezionale non si ha la garanzia che la prima soluzione fornita sia quella ottima. Si rende per tale motivo necessaria l'introduzione di una condizione di terminazione che assicuri il conseguimento dell'ottimo. Tale condizione prevede di sviluppare tutti i percorsi nelle due frontiere che abbiano un costo stimato da *f(n)* inferiore alla lunghezza della soluzione trovata.

# 3 Euristiche

Nel programma sono disponibili 3 diverse euristiche: il numero di celle collocate in modo errato, la distanza Manhattan e quest'ultima combinata con i Linear Conflicts.

### 3.1 Manhattan Distance

È realizzata semplicemente calcolando il numero di righe e di colonne di differenza tra la posizione di una cella in un certo stato e quella nello stato goal. Dato che gli stati sono memorizzati in tuple sequenziali e non in matrici si ricorre alle proprietà degli operatori // (divisione intera) e % (modulo) di python per ricavare gli indici di riga e colonna delle varie celle.

### 3.2 Manhattan Distance + Linear Conflicts

In questa euristica alla distanza Manhattan si aggiungono punti di distanza per ogni conflitto lineare: due celle $t_j$ e $t_k$ si dicono in conflitto lineare se sono collocate sulla stessa riga, le posizioni di ambo $t_j$ e $t_k$ nello stato goal si trovano sulla stessa riga, $t_j$ è a sinistra di $t_k$ e nella posizione finale invece deve trovarsi a destra (o viceversa). Allo stesso modo è definito il conflitto lineare sulle colonne. Per ogni conflitto vengono aggiunti due punti di distanza perché per risolverlo sarà necessario spostare una cella in un'altra riga (1 mossa), se il conflitto è sulle righe, per far passare le altre tessere e poi riportarla alla posizione precedente (1 mossa). L'implementazione dell'algoritmo per il calcolo dell'euristica sopra descritta si basa sullo pseudocodice fornito in (Hansson et al. 1992) a cui si aggiungono le funzioni conflict_rows() e conflict_columns() per il calcolo dei conflitti rispettivamente in ogni riga e colonna. Si analizza esclusivamente il primo in quanto i due metodi risultano equivalenti. Vengono definiti un contatore ed una lista in cui vengono riportate le tessere in conflitto con quella in questione. Per trovarle si controlla che la cella in questione sia nella riga corretta e per ogni elemento della riga si controlla che se anche altri elementi sono collocati correttamente essi siano già nell'ordine previsto, altrimenti la cella viene inserita nella lista dei conflitti ed incrementato il contatore.

## 4 Analisi dei risultati ottenuti

### 4.1 Analisi delle euristiche

Il primo test che è stato eseguito è quello volto all'analisi delle diverse euristiche. L'algoritmo A* è stato eseguito usando le tre diverse euristiche: il semplice numero di tessere collocate in modo errato, la distanza Manhattan e quest'ultima combinata con i Conflitti Lineari.
Il test prevede l'esecuzione dell'algoritmo con le 3 diverse euristiche per 100 volte su un stato input che è stato ottenuto da uno shuffle casuale dello stato iniziale di un numero variabile (da 20 a 29) di mosse. I risultati ottenuti come media del numero di stati esplorati in base al numero di shuffle in input sono mostrati nel grafico in fig. 1. Come si evince chiaramente dal grafico, l'euristica peggiore è quella che conta il numero di celle essendo la più lontana dalla reale distanza dello stato considerato da quello finale. Questo infatti provoca la necessità di espandere un numero di stati molto superiore rallentando molto l'esecuzione.

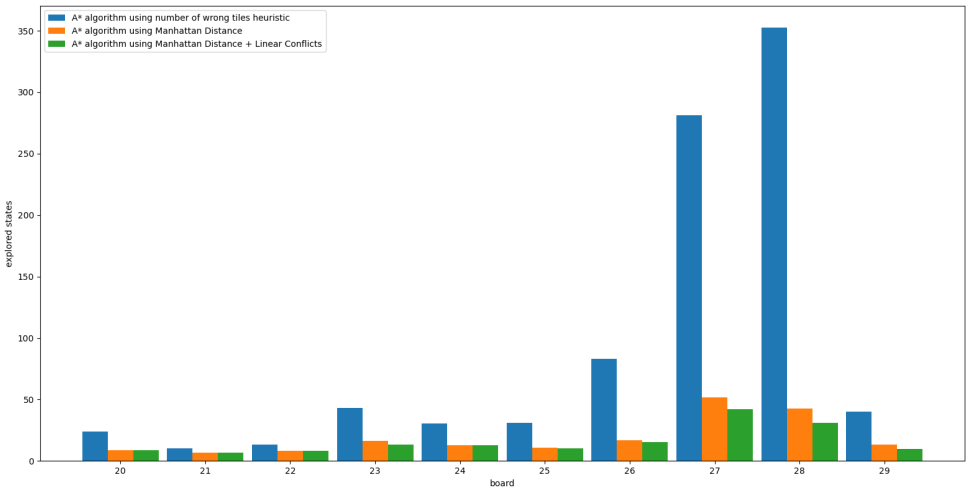


Figura 1: Il grafico a barre mostra il numero di stati esplorati dall'algoritmo A* monodirezionale usando le diverse euristiche per input soggetti a shuffle di 20-29 tessere

Per quanto riguarda invece le altre, i risultati forniti sono più vicini, in quanto i conflitti lineari migliorano leggermente l'approssimazione della distanza. Tale miglioramento risulta più evidente all'aumentare del numero di shuffle dell'input in quanto maggiore è la distanza dallo stato goal e più frequenti risultano i conflitti lineari. In generale si può quindi osservare che la combinazione delle due euristiche garantisce un'individuazione della soluzione più rapida in quanto l'euristica combinata domina la distanza Manhattan garantendo che l'algoritmo A* non espanderà mai più nodi rispetto al caso della sola distanza. Lo stesso esperimento è stato condotto nel caso di A* bidirezionale ed i risultati ottenuti mostrati nel grafico in fig. 2. Anche in questo caso, pur essendo meno marcata la differenza, l'euristica peggiore rimane il numero di tessere collocate in modo errato, seguita dalla Distanza Manhattan e poi dalla sua combinazione con i Conflitti Lineari.

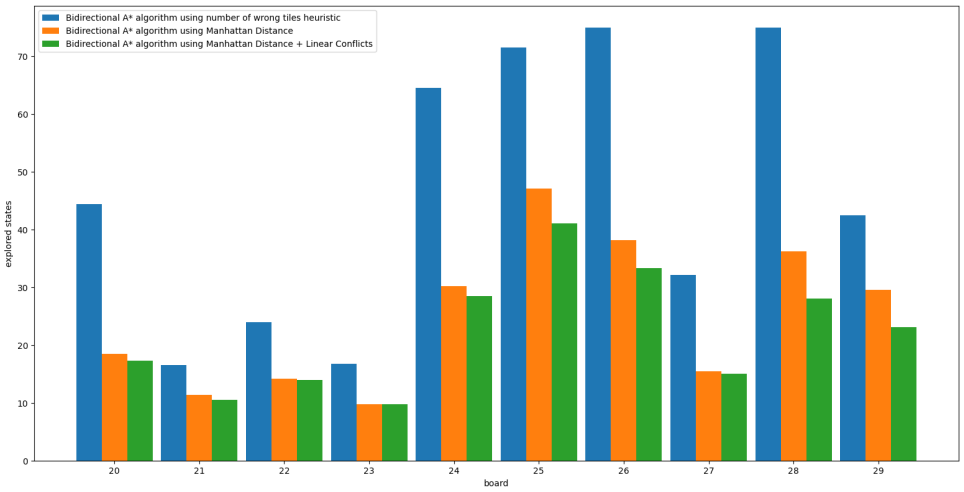


Figura 2: Il grafico a barre mostra il numero di stati esplorati dall'algoritmo A* bidirezionale usando le diverse euristiche per input soggetti a shuffle di 20-29 tessere

## 4.2 Analisi di A* star

Per quanto riguarda l'analisi di A* nelle forme monodirezionale e bidirezionale i test effettuati prevedono sia la verifica dell'ottimalità di entrambi gli algoritmi sia le prestazioni in termini del numero di stati esplorati. I test prevedono l'uso della stessa euristica, applicando i due algoritmi per 100 volte ad uno stato ottenuto facendo uno shuffle di un numero variabile di mosse dello stato goal. Il grafico in fig. 3 confronta il numero medio di stati esplorati dai due algoritmi. In entrambi i casi si può osservare un andamento esponenziale a mano a mano che lo stato in input si allontana dal goal, ma nel caso unidirezionale la crescita esponenziale presenta un coefficiente di crescita maggiore rispetto al caso bidirezionale come era lecito attendersi. Il secondo grafico della figura mostra invece la lunghezza della soluzione fornita dai due algoritmi. Questo grafico è unicamente fornito per dare ulteriore prova dell'ottimalità della soluzione ottenuta in quanto i due algoritmi forniscono sempre soluzioni della medesima lunghezza con crescita pressochè lineare all'aumentare del numero di shuffle delle tessere.

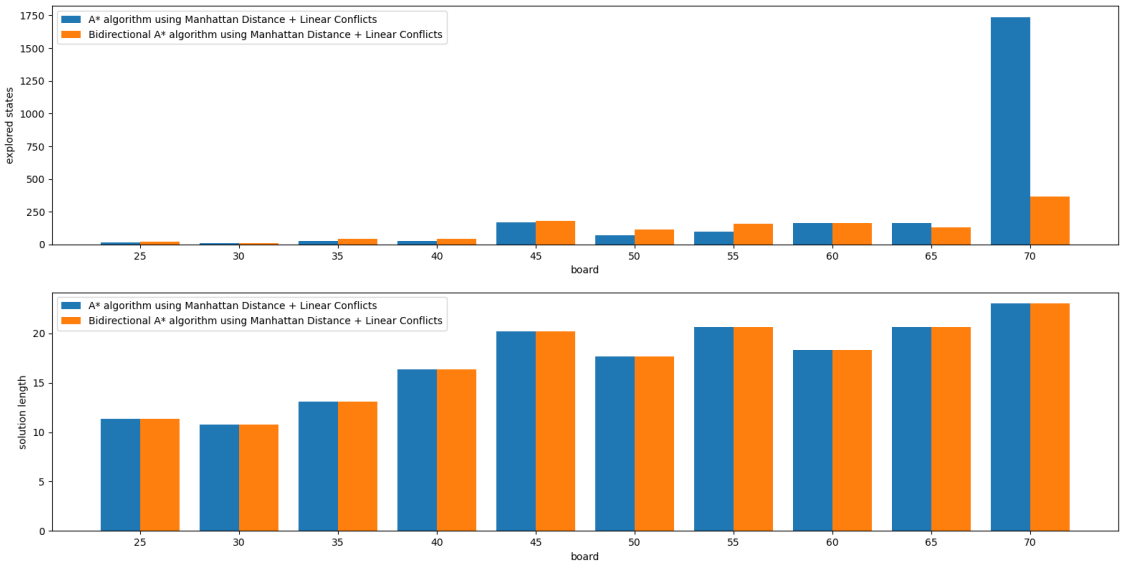


Figura 3: Il primo grafico analizza il numero di stati esplorati nei 2 casi; il secondo mostra invece la lunghezza della soluzione fornita dai 2 algoritmi.

Si fornisce infine una comparazione delle 4 versioni dell'algoritmo nel grafico di fig. 4. Anche in questo caso è osservabile l'andamento esponenziale con crescita molto più rapida nel caso unidirezionale rispetto a quello bidirezionale (l'andamento atteso era rispettivamente $b^d$ e $2b^{d/2}$ e con una sempre più apprezzabile differenza tra le due euristiche a mano a mano che l'input si allontana dal goal visto che i conflitti tra le tessere si verificano più frequentemente.

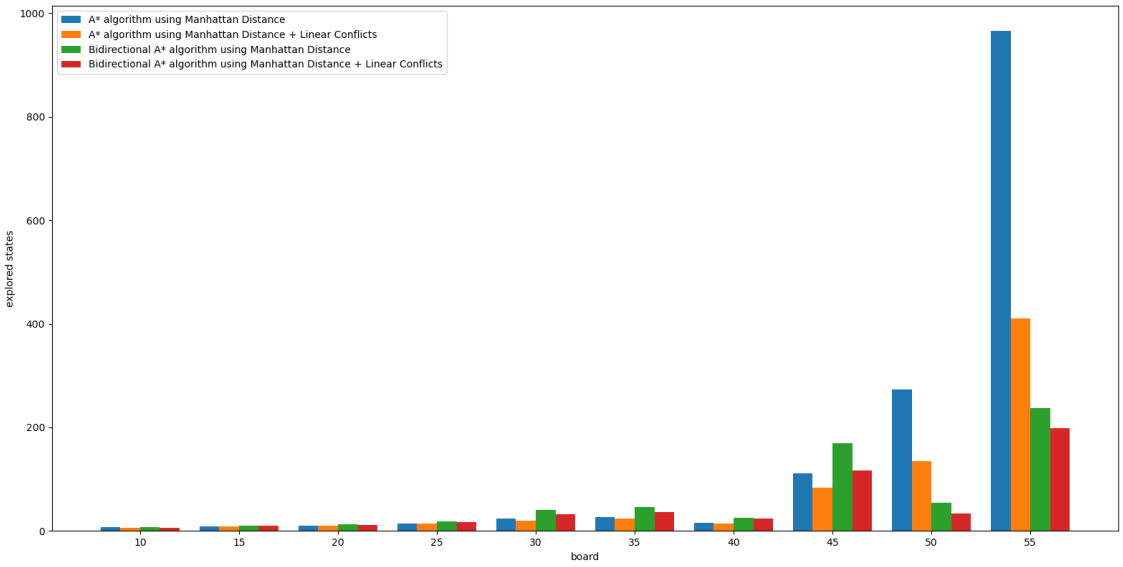


Figura 4: Il grafico analizza il numero di stati esplorati nei 4 casi.