Relazione elaborato di Ricerca Operativa

Pietro Tellarini

Agosto 2023

1 Introduzione

Come elaborato di Ricerca Operativa ho scelto di implementare l'algoritmo A-star (o A*), un algoritmo di ricerca di percorsi in un grafo. È un algoritmo deterministico, il che significa che trova sempre una soluzione al problema, se esiste, ed è anche un algoritmo completo, il che significa che trova la soluzione migliore possibile, in termini di lunghezza del percorso.

Tale algoritmo è di natura euristica, ovvero utilizza una funzione di "valutazione" per guidare la ricerca tra i nodi possibili del grafo esaminato e funziona tenendo traccia del percorso più breve trovato da ogni nodo. Inizia dal nodo di partenza e, per ogni nodo successivo, aggiorna il percorso più breve se il nuovo percorso è più breve del percorso già conosciuto, continuando in questo modo fino a raggiungere il nodo di destinazione.

L'algoritmo A* è ampiamente utilizzato in applicazioni di navigazione, robotica e ottimizzazione. Ad esempio, è utilizzato nei sistemi di navigazione GPS per trovare il percorso più breve tra due punti. È anche utilizzato nei robot per pianificare i movimenti e nelle applicazioni di ottimizzazione per individuare la soluzione migliore a un problema.

2 Algoritmo A*

Nello specifico l'algoritmo A* è un algoritmo euristico perché utilizza la funzione di valutazione del costo del percorso (costo totale) per guidare la sua ricerca. La funzione di costo di percorso è una stima della lunghezza del percorso più breve dal nodo corrente al nodo di destinazione.

Gli algoritmi euristici sono spesso più efficienti degli algoritmi non euristici, perché possono escludere rapidamente i nodi che non hanno la probailità di condurre ad una una soluzione ottimale. Tuttavia, gli algoritmi euristici non possono garantire di trovare la soluzione ottimale ma solo una che è la migliore possibile in base alla funzione di valutazione utilizzata.

Nel caso dell'algoritmo A*, la funzione di costo di percorso può essere imprecisa, specialmente se il grafo è complesso, anche se nella maggior parte dei casi si è comunque in grado di trovare una soluzione ottimale.

2.1 Funzione di valutazione

L'algoritmo trova il percorso ottimale per raggiungere la soluzione scelta tramite l'uso di una funzione di valutazione che determina un costo totale in base a quanto il nodo processato sia vicino alla soluzione desiderata.

Tale funzione può essere quindi utilizzata per esplorare i nodi del grafo in ordine crescente di costo totale. Il costo totale viene definito dalla seguente formula:

$$cost = depth + h \tag{1}$$

Dove *depth* rappresenta la profondità in cui si trova il nodo nell'albero mentre h determina il coefficiente euristico.

2.2 Coefficiente euristico (h)

Nel caso proposto si è scelto di utilizzare due possibilità per il calcolo del coefficiente euristico. Nello specifico le implementazioni che sono state proposte sono quelle del numero di tessere nella posizione errata e quello della distanza di Manhattan.

2.2.1 Wrong Position

Indica esclusvamente il numero di tessere che si trovano nella positione errata rispetto alla soluzione, al momento della valutazione.

2.2.2 Manahattan distance

La distanza di Manhattan è una misura della distanza tra due punti in un piano, misurata lungo gli assi x e y. È spesso utilizzata in applicazioni che si occupano di muovere oggetti in una griglia, come i videogiochi.

$$h = |x_1 - x_2| + |y_1 - y_2| \tag{2}$$

Dove x_1 e y_1 reppresentano le coordinate del piano cartesiano dove si trova la soluzione, mentre x_2 e y_2 reppresentano le coordinate reali attuali di tale punto nel piano.

2.3 Pseudocodice

A* consiste quindi nell'esplorare ad ogni ciclo il nodo con costo minore (distanza minore dalla soluzione), valutare se esso si tratti del nodo obiettivo ed infine generare i successori di tale nodo calcolandone i relativi costi totali.

```
function astar(puzzle):
  # Inizializza la coda di ricerca con il nodo iniziale.
  open = [initial_node]
  # Inizializza un set che contiene i nodi già esplorati.
  visited = set()
  # Finché la coda di ricerca non è vuota:
  while open(open) > 0:

    # Estrai il nodo con il costo totale più basso dalla coda di ricerca.
    node = open.pop(0)
    # Se il nodo è l'obiettivo:
    if node == goal:
      return node.path
    # Aggiungi i nodi filgi alla coda di ricerca.
    open.put(node.get_children())
    # Aggiungi il nodo al set di nodi già esplorati.
    visited.add(node)

  # Il puzzle non ha una soluzione.
  return None
```

Si sottolinei come nell'implementazione si sia deciso di creare una coda con priorità per i nodi da esplorare, ridefinendo l'operatore di minore per gli oggetti nodo. In questo modo si è velocizzata l'estrazione del nodo con costo più basso durante l'esecuzione dell'algoritmo.

3 8 Puzzle

L'elaborato si pone l'obiettivo di creare un risolutore per il gioco 8 Puzzle che implementi l'algoritmo A-star in maniera efficiente e veloce, trovando una soluzione ammissibile e gli step per raggiungerla, dato un problema nel dominio del gioco.

3.1 Il gioco

8 Puzzle consiste in una griglia 3 per 3 nel quale sono presenti otto tessere numerate da 1 a 8 e nel quale rimane uno spazio vuoto. Lo spazio vuoto consente di poter muovere le altre tessere adiacenti in modo da poter raggiungere la soluzione finale, che si ottiene ordinando le tessere da 1 a 8, seguendo l'ordinamento per riga, e lasciando vuota la casella nell'angolo in basso a destra (figura: 1).

3.2 8 Puzzle visto come un grafo

Nel problema in questione si intende modellare il puzzle come un grafo ad albero in cui, dal nodo iniziale, si possano generare i nodi figli fino al raggiung-

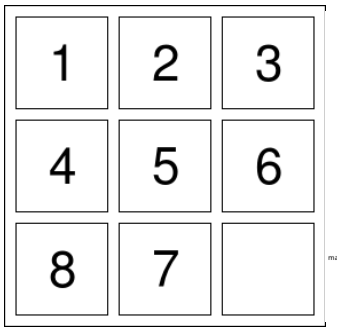

Figure 1: Puzzle Example

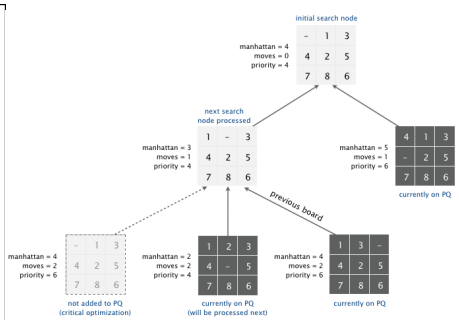

Figure 2: Puzzle as a graph

imento dell'obiettivo.

Nello specifico si è scelto di considerare come nodi gli stati in cui la griglia si può trovare, considerando l'ordine delle tessere. I nodi figli vengono generati dal predecessore tramite i movimenti che le tessere adiacenti allo spazio vuoto possono compiere. Ne consegue che, per la risoluzione del problema, tra i figli generati dal nodo processato, non venga considerato il suo nodo padre; ciò infatti comporterebbe la creazione di un nodo duplicato che è già stato esplorato e con distanza maggiore perchè si tratterebbe di effettuare un passo a ritroso nell'albero del problema. (figura: 2).

4 Applicativo

L'applicativo è stato sviluppato in python, non presenta alcuna interfaccia grafica ma il tutto viene gestito da riga di comando.

L'utilizzo è semplice, basta avviare l'applicativo con il comando *python A-Star.py*. Da terminale verrà richiesto di inserire un caso da risolvere. E' sufficiente scrivere i numeri in riga lasciando lo spazio tra i valori.

Verrà successivamente chiesto come si vuole calcolare la distanza euristica altrimenti di default verrà scelta la distanza di Manhattan (solitamente più efficiente).

L'applicativo mostrerà infine il tempo impegato per la risoluzione e le mosse da effetuare per arrivare, dallo stato iniziale impostato, alla soluzione.

4.1 Testing

L'applicativo è stato testato su vari casi di esempio:

Caso d'esempio	Manhattan (sec.)	WrongPosition (sec.)	mosse sol.
1 2 3 4 5 6 7 8 0	0.0	0.0	0
4 1 3 7 2 6 0 5 8	0.0	0.0	6
7 2 4 5 0 6 8 3 1	0.0	0.0351	20
1 4 3 6 5 8 2 7 0	0.015	0.087	22
8 7 6 5 4 3 2 1 0	0.123	1.428	30
6 4 7 8 5 0 3 2 1	0.083	1.883	31
8 6 7 2 5 4 3 0 1	0.085	1.884	31

I tempi dei nodi possono essere soggetti a lievi variazioni in quanto non è stato utilizzato un campione sufficientemente grande per calcolare un valor medio valido.

4.2 Non tutto è risolvibile

Non tutte le combinazioni possibili dell'8 Puzzle hanno una soluzione. Alcune configurazioni dell'8 Puzzle sono insolubili, il che significa che non possono essere risolte per raggiungere lo stato di obiettivo, indipendentemente da quante mosse vengano fatte.

Per capire quali combinazioni sono insolubili, si può considerare l'ordine delle tessere nella configurazione iniziale: se la disposizione delle tessere ha un numero pari di inversioni (ovvero il numero di volte che una tessera precedente è posizionata dopo una tessera successiva), allora la configurazione è considerata insolubile, tuttavia, se ha un numero dispari di inversioni, potrebbe avere una soluzione.

Questa regola si basa sulla parità delle permutazioni: in una configurazione risolvibile, si può sempre scambiare una tessera con la tessera vuota per raggiungere la configurazione obiettivo. Nei casi insolubili, a causa delle permutazioni pari, non è possibile scambiare le tessere in modo che la configurazione obiettivo sia raggiungibile.