UNIVERSITÀ DEGLI STUDI DI FIRENZE, INGEGNERIA INFORMATICA

Guida di robots

Emanuele Nencioni

February 5, 2021

1 DESCRIZIONE DEL PROBLEMA

1.1 Analisi

L'esercizio richiede di trovare il percorso più breve tra due punti su un piano che ha ostacoli poligonali convessi, come mostrato in figura 3.1. Un' idealizzazione di un robot che attraversa un ambiente affollato.

1.2 Formalizzazione del problema

Per prima cosa occorre trovare lo spazio degli stati. Supponiamo che questo sia formato da tutte le posizioni (x, y) del piano, allora avremo che la dimensione di questo spazio è praticamente infinita, a meno che non si approssimi i valori delle posizioni a semplici interi; ma anche in questo caso, il numero delle posizioni sarebbe comunque elevato e renderebbe intrattabile la navigazione del robot all'interno di ambienti di grandi dimensioni. Una soluzione alternativa e migliore, è quella di utilizzare come spazio degli stati, la posizione (x, y) dei punti di start, goal e dei vertici dei poligoni. In questo modo riusciremo ad ottenere uno spazio degli stati con dimensione nettamente minore, che dipende dal numero di poligoni e da quanti lati hanno. Supponiamo che i poligoni abbiano tutti al massimo d lati, allora, nel caso peggiore, avremo uno spazio degli stati dell'ordine di

$$O(2+nd)$$

con n il numero dei poligoni e 2 rappresenta lo stato iniziale e finale. Il percorso più breve per andare da un vertice ad un'altro è proprio una linea retta tra i due; perciò si potrebbe utilizzare un segmento che colleghi un vertice con i vertici dello stesso poligono adiacenti, o vertici di altri poligoni che non passano attraverso i poligoni a cui appartengono o altri(perché significherebbe che la strada percorsa dal robot passerebbe attraverso un ostacolo). Quindi le azioni, saranno appunto delineate, da tutti questi possibili segmenti di collegamento tra vertici/punti. Il modello di transizione, indicherà quale sarà il nuovo punto indicato dall'azione che corrisponderà al nuovo stato. Per verificare lo stato di goal, basterà confrontarlo con quello a cui si trova il programma. Per quanto riguarda il costo del cammino, questo equivale alla somma dei costi dei segmenti, quindi alla somma delle lunghezze dei segmenti.

2 DESCRIZIONE DEL CODICE

2.1 Strutture dati

Per utilizzare tutti gli algoritmi a dovere, ho creato una struttura dati di un grafo creando la classe nodo. Questa, ha al suo interno, un campo per indicare: lo stato, il padre, l'azione eseguita dal padre per ottenere quel nodo, e il costo del cammino fino a quel nodo. La classe è poi fornita di tutte le funzioni necessarie per esplorare il grafo e ritornare la soluzione. Per quanto riguarda tutto quello che comprende la parte geometrica dell'esercizio, ho usato la libreria **scikit-geometry**. Inoltre il codice del file *utils.py* insieme al codice di due algoritmi, è stato preso dalla repository `https://github.com/aimacode`.
Ho creato poi la classe che definisce e formalizza il problema: **RobotRoute**, che ha la stessa interfaccia di un problema di ricerca standard:

- **__init__**(state, goal_state, map) inizializza il problema.
- **action**(state) ritorna le azioni disponibili per quello stato.
- **result**(action, state) ritorna un nuovo stato, partendo dallo stato state applicando l'azione action.
- **goal_test**(state) controlla se lo stato attuale è quello di goal.
- **path_cost**(state, action) tiene traccia di tutto il costo da start fino al nodo attuale.
- **h**(node) definisce una euristica per il problema. In questo caso si usa la distanza Euclidea dal nodo dato allo stato di goal.

2.2 Trovare le azioni di uno stato

Per trovare tutte le azioni di uno stato, la funzione **action**(state) utilizza due funzioni extra private:

- **__segment_of_pol**(*point, poly*), ritorna tutti i possibili segmenti che partono da *point*, verso il poligono *poly*, senza che questi si intersechino con il poligono stesso.
- **__intersection_pol**(*seg, poly*) , dato un segmento *seg*, ritorna true se questo interseca/-passa attraverso il poligono *poly*, altrimenti false.

La funzione **action**(*state*) quindi, per ogni poligono, mappa tutti i vertici a cui è possibile arrivare dallo stato state. Dopodiché controlla se i segmenti non passino attraverso altri poligoni; controlla pure se esiste un segmento, che dallo stato attuale, possa arrivare allo stato di goal ed infine, ritorna la lista di tutti i segmenti che rispettano queste regole.

2.3 Algoritmi di ricerca

Sono stati usati 3 algoritmi di ricerca:

2.3.1 Algoritmo Breadth first search(BFS)

Questo algoritmo effettua una visita in ampiezza del grafo dei nodi, avendo come frontiera una struttura dati di tipo FIFO. Si utilizza la variante *graph-search*, la quale ha una lista di nodi chiamata *explored* che permette di non espandere quelli già esplorati e quindi di risparmiare tempo. L'algoritmo in questione ha un costo in termini di tempo e spazio equivale a $O(b^{d+1})$, dove b è il branching factor, d è la profondita.

2.3.2 Algoritmo Uniform cost search(UCS)

Algoritmo in cui la struttura dati della frontiera è una coda con priorità, guidata dal costo minimo, quindi $f(n) = g(n)$dove n è il nodo, $g(n)$ è il costo del cammino dallo stato iniziale fino a

quel nodo e $f(n)$ è detta funzione di valutazione. Il codice è stato preso dalla repository https://github.com/aimacode così come il codice di *A* search*: si utilizza l'algoritmo generico di **Best first search** (anche qui si usa la variante *graph-search*) che, a seconda di come è fatta $f(n)$, cambia il suo comportamento. Il costo spaziale/temporale è,nel caso peggiore, un $O(b^{1+\lfloor \frac{C*}{\epsilon} \rfloor})$, dove $C*$ è il costo di un cammino per una soluzione ottima, e ϵ è il costo per un singolo step-cost (costo per andare da uno stato ad un altro).

2.3.3 Algoritmo A search*

Algoritmo molto simile a *UCS* (poiché A* non fa altro che una chiamata allo stesso algoritmo che usa UCS: **Best first search**), ma in questo caso,la frontiera, è guidata da una priorità del tipo:

$$f(n) = g(n) + h(n)$$

dove $h(n)$ è una funzione euristica. In questo caso specifico, come già detto, $h(n)$ è la distanza in linea retta del nodo n allo stato di goal, che corrisponde alla distanza euclidea. Si utilizza sempre la versione *graph-search*. Nel caso peggiore, il costo è un $O(b^d)$ sia temporale che spaziale come *BFS*, ma, dato che segue una euristica, se questa è consistente, aiuterà l'algoritmo ad "instradarsi" verso la soluzione esplorando per primi i nodi che più si avvicinano allo stato di goal.

3 Esperimenti svolti

3.1 Mappe usate

le mappe usate per fare i test sono due. Il programma creerà le mappe direttamente dal codice.

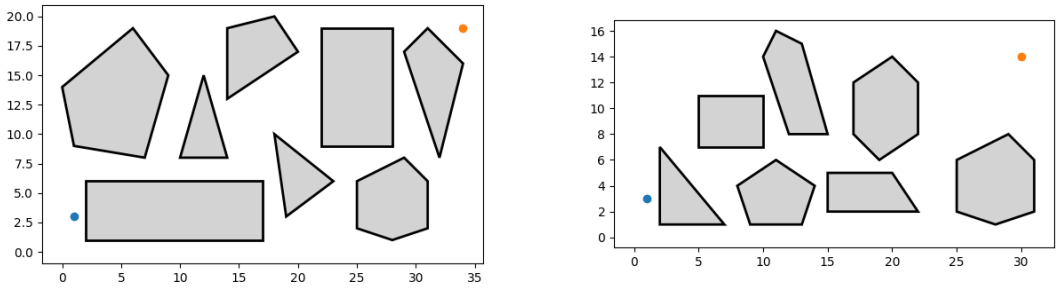

Figure 3.1: A sinistra la mappa 1, a destra la mappa 2

3.2 Risultati esperimenti

3.3 Analisi risultati

I risultati sono stati fatti tramite il file *test.py*, utilizzando una media di 10 esecuzioni. Nelle tabelle 3.1 e 3.2, possiamo vedere che il vincitore in entrambi i casi è *A**, il quale riesce a trovare il cammino con costo minimo e in minor tempo possibile. Questo soprattutto perché

Algoritmo	**tempo(s)**	**Costo cammino**
Breadth first search	0.25936	46.9284
Uniform cost search	4.16558	40.92927
A* search	0.09837	40.92927

Table 3.1: Prestazioni su mappa 1

Algoritmo	**tempo(s)**	**Costo cammino**
Breadth first search	0.40926	35.70947
Uniform cost search	19.86316	34.75804
A* search	0.07210	34.75804

Table 3.2: Prestazioni su mappa 2

la distanza euclidea è un'euristica consistente e rende l'algoritmo ottimo. Per quanto riguarda l'algoritmo, *BFS*, non avendo nessun controllo sul costo minimo, la sua ricerca si ferma sul primo cammino che porta allo stato di goal(Figura 3.2). In questo caso non è il migliore. Ovviamente se la scelta dell'ordine delle posizioni dei vertici dalla mappa fosse causale, *BFS* mostrerebbe cammini sempre diversi.

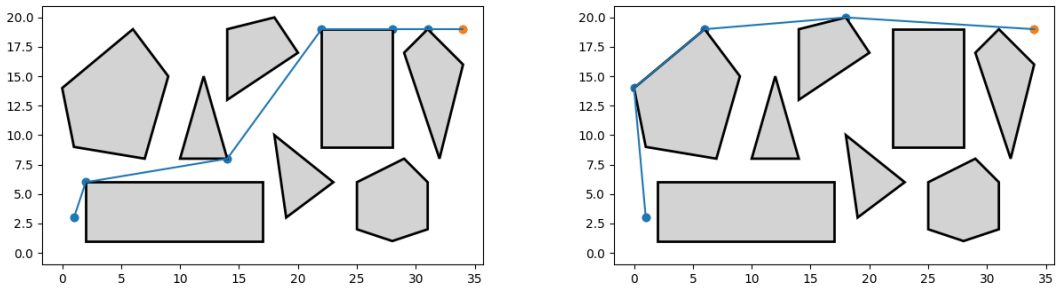

Figure 3.2: A sinistra cammino di *A*/UCS*, a destra il cammino di *BFS*

Resta il fatto che, siccome lo spazio degli stati è molto piccolo e lo stato di goal è raggiungibile quasi subito da stati molto vicini a quello iniziale,*BFS* rimane in seconda posizione per velocità, ma risulta un algoritmo non ottimo. L'algoritmo *UCS*, si trova ultimo in termini di tempo, poiché cerca sempre di mantenere il costo uniforme sulla frontiera, quindi troverà il cammino che porta al goal, solamente quando il costo della frontiera equivalerà al costo minimo per arrivare al goal, perciò è un algoritmo ottimo. A differenza di *A**(che di fatto si basa sullo stesso codice di *UCS*), non ha una euristica che "instradi" l'algoritmo nei percorsi migliori, perciò, espanderà tutti i nodi sulla frontiera anziché quei pochi che porterebbero più vicino alla soluzione, quindi risulta molto più lento.