UNIVERSITÀ DEGLI STUDI
DI PERUGIA

Tesina di

Algoritmi e Strutture di dati

Corso di Laurea in Ingegneria Informatica ed Elettronica – A.A. 2020-2021

Dipartimento di Ingegneria

docente

Prof. Emilio Di Giacomo

Implementazione e analisi dell'algoritmo A* per problemi di ricerca del cammino minimo

studenti

316649	**Francesca**	**Nocentini**	francesca.nocentini@studenti.unipg.it
312294	**Paolo**	**Speziali**	paolo.speziali@studenti.unipg.it

Data ultimo aggiornamento: 9 giugno 2021

0. Indice

1. Problema Affrontato

Il progetto si basa sull'implementazione dell'algoritmo di ricerca A* in linguaggio Javascript V8.

Si tratta di un algoritmo di tipo greedy best-search, cioè utilizza un'euristica che permette di stimare, data una rete di percorsi, quello che è il cammino migliore verso un determinato traguardo. Tale algoritmo si avvale dell'utilizzo di grafi pesati (con pesi non negativi) e lo scopo è quello di partire da un vertice di partenza (*start*) per giungere a un vertice di arrivo (*goal*) impiegando il minor costo totale.

Nell'ambito delle problematiche di ricerca dei cammini minimi il primo pensiero può andare verso una soluzione implementata con il classico algoritmo di Dijkstra o con altri che svolgono una ricerca in ampiezza. Tuttavia la conoscenza di maggiori informazioni, nel nostro caso la posizione del punto di arrivo, ci permette di sfruttare questa soluzione decisamente meno onerosa dal punto di vista computazionale.

L'algoritmo A* può essere considerato un algoritmo analogo a quello di Dijkstra con la differenza che, mentre quest'ultimo esplora tutti i possibili cammini, tra i nodi connessi, con un raggio d'azione circolare che va espandendosi fino a trovare il traguardo e su questi definisce quale è il migliore, A* cerca direttamente il cammino minimo usando la funzione euristica e quindi dando priorità ai nodi che sono stimati essere migliori di altri. Ovviamente A* funziona bene solo quando l'euristica che viene definita per i nodi è ammissibile, cioè non sovrastima mai la distanza effettiva verso la meta. Nel nostro caso come euristica ammissibile abbiamo utilizzato la distanza effettiva in linea d'aria tra i nodi e il goal.

A* funziona in maniera ottimale se l'euristica è anche consistente, cioè tale che per ogni nodo N e per ogni suo vicino P del grafo risulta che

$$h(N) \leq h(P) + c(N, P)$$

dove c(N,P) è il peso del grafo tra N e P.

La nostra euristica soddisfa tale proprietà in quanto è possibile costruire sempre un triangolo con i vertici N, P e il goal G e, sapendo che la lunghezza del lato NP è sempre minore della somma degli altri due, tale uguaglianza vale.

Possiamo dire che A* è sicuramente più efficace ed efficiente rispetto a Dijkstra, per grafi molto ampi, se si vuole calcolare un percorso ottimo tra due punti in maniera rapida, a scapito ovviamente di un maggiore utilizzo della memoria dovuto alla presenza dell'euristica. Se si hanno a disposizione le informazioni che ci permettono di definire l'euristica, A* è preferibile.

Dijkstra è sicuramente migliore per problemi di pathfinding in quanto viene trovato, se esiste, il percorso migliore, confrontandone molti tra loro, a discapito della rapidità con cui questo viene elaborato.

2. Algoritmi implementati e strutture dati realizzate

L'algoritmo di A* vero e proprio è stato inserito in un modulo NodeJS da cui poter richiamare il metodo $astar(start, goal, graph)$, il quale prende come parametri due oggetti di classe Vertex start e goal e un oggetto di classe Graph.

Le due classi appena citate sono state realizzate come semplici implementazioni di questi concetti (vertice e grafo), Graph avente una lista di adiacenza e i metodi per gestirla e Vertex avente degli attributi per l'id del nodo e le coordinate x e y e il metodo per il calcolo della funzione euristica che permette di calcolare la distanza euclidea tra il vertice su cui viene chiamato tale metodo e un altro passato per riferimento.

Ad ogni iterazione del ciclo dominante della nostra funzione $astar$, per ogni vertice preso in esame, si decide quale arco attraversare sulla base del costo di esso e della stima formulata in vista del raggiungimento di quel determinato goal. Viene in particolare scelto quel nodo che minimizza il valore fScore f(n) tale che

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

dove n è uno dei vertici vicini a quello che è stato elaborato fino a quel momento, g(n), anche detto gScore, rappresenta il costo del percorso dal nodo di start al nodo n e h(n) è invece l'euristica ammissibile che stima il costo minore sulla sola base del raggiungimento del goal (calcolata tramite il metodo della classe Vertex sopracitato). Si esce dal ciclo dominante una volta che è stato raggiunto il goal e, in seguito, viene ricostruito il percorso andando a ritroso, partendo dal goal e tenendo traccia per ogni nodo del suo predecessore, finché esso non sarà il nodo di start.

Per implementare l'algoritmo sono state utilizzate principalmente quattro strutture dati:

- **openSet**: l'insieme dei vertici che sono stati scoperti, e dei quali ancora occorre scoprire i vicini, e dei vertici già analizzati in precedenza per cui si tenta però di trovare un percorso di costo minore (eventualità rara se si utilizza una funzione euristica consistente come nel nostro caso);
- **cameFrom**: l'insieme dei vertici candidati per essere inclusi nel percorso finale associati al nodo da cui si proviene per arrivarci;
- **gScore**: contiene tutti i vertici con associato il percorso più economico conosciuto fino a quel momento per arrivare ad esso a partire dal vertice di start;
- **fScore**: presenta tutti i vertici con associato il minimo valore di f(n), ciò rappresenta il costo del percorso stimato più breve fino a quel momento per arrivare al goal passando attraverso il nodo n.

Come primo passo abbiamo inizializzato tutti i gScore ed fScore con il valore Infinity (confrontabile con altri tipi di dato) ed openSet contenente solo il nodo di start.

Per partire con l'analisi dei vertici si entra in un ciclo che si conclude solamente o quando estraiamo il goal da openSet o quando sempre openSet risulta vuoto. Infatti, tra i vertici contenuti in questa struttura dati, prendiamo all'inizio di ogni ciclo quello con il valore di fScore minimo in quel momento e lo rimuoviamo.

Sempre nella stessa iterazione openSet verrà popolato dei vicini del vertice appena rimosso, in caso essi non siano già presenti, nell'evenienza che quest'ultimi abbiano un valore di gScore per il percorso attualmente in esame minore rispetto a quello precedentemente riscontrato (aggiornando ovviamente i valori di gScore, fScore e cameFrom ad essi associati). L'avvenimento di un openSet vuoto indica un fallimento nel trovare un percorso verso il goal.

Possiamo considerare un successo invece l'evento in cui il nodo di openSet con il minimo valore in fScore è proprio il goal. A quel punto viene richiamato il metodo reconstructPath che partendo dal nodo di goal procede al contrario e ricostruisce il percorso sulla base della lista di predecessori cameFrom.

Algorithm 1: Iterazione principale dell'Algoritmo A*

```
while openSet non è vuoto do
    current = il nodo in openSet con il valore di fScore[] più basso;
    if current == goal then
        return reconstructPath(cameFrom, current);
    end
    openSet.Remove(current);
    for each neighbor of current do
        gScoreTentative = gScore[current] + d(current, neighbor);
        if gScoreTentative < gScore[neighbor] then
            cameFrom[neighbor] = current;
            gScore[neighbor] = gScoreTentative;
            fScore[neighbor] = gScore[neighbor] + h(neighbor);
            if neighbor not in openSet then
                openSet.add(neighbor);
            end
        end
    end
end
```

Le strutture dati utilizzate in prima istanza per i set necessari sono state un Array dinamico per openSet e delle Map per gli altri tre, in quanto si trattano di strutture dati dinamiche costituite da coppie di variabili che ci permettono facilmente di associare ogni vertice ad un valore, che sia l'id del nodo predecessore per cameFrom, che sia il cammino di costo minimo attualmente conosciuto per gScore o che sia il valore minimo di f(n) per fScore.

La struttura del grafo è stata implementata tramite l'utilizzo di liste di adiacenza, in quanto non è tanto importante sapere se due vertici sono vicini, quanto conoscere quali siano i vicini di un dato vertice, dunque l'utilizzo di matrici di adiacenza non avrebbe portato particolari benefici.

Nel caso di openSet, al fine di evitare di introdurre un fattore di complessità O(m) (con m numero di vertici in openSet) ogni volta che occorreva cercare il vertice con il minimo valore di fScore associato, abbiamo deciso di cambiare la sua struttura da semplice array ad una lista di priorità, realizzata con un min-heap binario, ordinato in senso crescente sulla base dei corrispettivi valori di fScore, così da avere, tramite i metodi per l'implementazione delle code, una complessità al più O(lgn). In questo caso invece di richiamare sui nodi adiacenti un metodo che effettua una ricerca del minimo, prendiamo direttamente il primo elemento dalla coda di priorità.

Il nostro min-heap binario è stato implementato con un Array *(NB: In Javascript la maggior parte degli Array sono delle strutture dinamiche paragonabili agli ArrayList di Java)*. Possiamo contare su un accesso diretto ai valori dell'fScore, perciò la Priority Queue ci permette di risparmiare sul tempo di elaborazione.

In molti pseudocodici dell'algoritmo reperiti online si utilizza anche un'altra struttura dati, closedSet, ovvero la lista dei nodi tolti da openSet e da non analizzare nuovamente se appaiono tra i vicini del nodo in analisi. Nel codice, per non andare ad aggiungere fattori di complessità, è stata implementata tramite una Map a cui all'ID del vertice viene associata una booleana. Tuttavia, testandola in una classe a parte, abbiamo notato che, grazie alla nostra funzione euristica consistente, i benefici a livello di tempo di esecuzione erano praticamente nulli anche per input più "impegnativi".

Per quanto riguarda la creazione dei grafi concreti su cui testare il nostro codice ci siamo basati su una convenzione definita dai file di dati della rete stradale californiana offerta dall'Università dello Utah: andiamo infatti a leggere riga per riga due file, uno in cui sono definiti i nodi del grafo con le loro coordinate spaziali e il loro ID, e uno in cui sono definiti gli archi, quali nodi comprendono e la distanza tra i due. Dalla lettura siamo in grado di utilizzare questi dati per popolare la lista di adiacenza del nostro grafo. I grafi presi in considerazione sono non orientati, quindi ovviamente l'applicazione non tiene conto delle direzioni reali lungo le quali ci si può muovere nella realtà o dei sensi unici.