Confronto tra gli algoritmi di Dijkstra e A per il Pathfinding su Reti Stradali

Silveri Marco - s330394
Sanna Emanuele – s330417

1. Introduzione

L'obiettivo di questo report è analizzare e confrontare due tra i più noti algoritmi di pathfinding: Dijkstra e A*. Entrambi sono ampiamente utilizzati per la ricerca del cammino minimo in grafi pesati, ma si distinguono per efficienza e strategia di esplorazione, specialmente quando applicati a problemi reali di navigazione.

Il contesto scelto per questa analisi è quello dei grafi pesati non direzionati, modellati tramite la libreria OSMnx di Python, che consente di scaricare e rappresentare porzioni di mappa reale (ad esempio una città) come grafo. In questo scenario, i nodi rappresentano o punti di interesse, mentre gli archi rappresentano le strade e sono pesati sulla base della distanza.

Il nostro obiettivo sarà quello di sviluppare un sistema che permetta di calcolare il cammino più breve tra due nodi della città (selezionati casualmente o in modo predefinito), in modo simile a quanto farebbe un'applicazione di navigazione come Google Maps. Ciò permetterà non solo di osservare il comportamento pratico di ciascun algoritmo, ma anche di valutarne le differenze in termini di performance e accuratezza.

2. Algoritmi

Nel confronto tra Dijkstra e A*, emergono differenze sostanziali legate soprattutto all'impiego **dell'euristica** e alla conseguente efficienza operativa. Dijkstra segue una logica puramente deterministica: esplora tutti i percorsi possibili partendo dal nodo sorgente, senza alcuna informazione preventiva sulla posizione della destinazione. Questo garantisce l'ottimalità della soluzione in ogni circostanza, ma può risultare inefficiente in presenza di grafi molto estesi.

A* introduce invece un elemento euristico, ossia una funzione di stima del costo rimanente per raggiungere il nodo obiettivo. Questa funzione guida la ricerca in maniera più intelligente, privilegiando i nodi che sembrano più vicini alla destinazione. Se l'euristica utilizzata è ammissibile (cioè non sovrastima mai il costo reale), A* mantiene l'ottimalità del risultato, riducendo al contempo i tempi di calcolo rispetto a Dijkstra. Questo lo rende più adatto per applicazioni come il pathfinding su mappe geografiche, dove la posizione della destinazione è nota e può essere sfruttata per ottimizzare l'esplorazione.

La qualità dell'euristica è determinante per le performance dell'algoritmo A*. In ambito spaziale, le più comuni sono:

- Distanza Euclidea: è l'euristica più naturale in uno spazio continuo, calcolata come distanza "in linea retta" tra due punti. Per questo motivo è stata scelta in questo lavoro.
- Distanza di Manhattan: è l'euristica più adatta a uno spazio a griglia ortogonale, calcolata come somma delle distanze verticale e orizzontale tra due punti.

3. Implementazione

https://github.com/emariesanna/TechnologiesForAutonomousVehicles

L’algoritmo di Dijkstra è implementato con una funzione che come prima cosa inizializza i nodi del grafo con distanze infinite, tranne il nodo di partenza. Utilizza una coda di priorità per esplorare i nodi in ordine di distanza crescente dal nodo iniziale, aggiornando i percorsi migliori trovati. Quando raggiunge il nodo di destinazione, ricostruisce il cammino minimo.

```python
def dijkstra(orig, dest, plot=False):
    for node in G.nodes:
        G.nodes[node]["visited"] = False
        # distanza complessiva tra il nodo di partenza e il nodo considerato
        G.nodes[node]["distance"] = float("inf")
        G.nodes[node]["previous"] = None
        G.nodes[node]["size"] = 0
    for edge in G.edges:
        style_unvisited_edge(edge)
    G.nodes[orig]["distance"] = 0
    G.nodes[orig]["size"] = 50
    G.nodes[dest]["size"] = 50
    pq = [(0, orig)] # pqueue: lista ordinata crescente con i nodi da visitare
    step = 0
    while pq:
        _, node = heapq.heappop(pq) # estrae il primo nodo di pq

        if node == dest:
            # print("Iterations:", step)
            #plot_graph()
            return step
        if G.nodes[node]["visited"]: continue # break
        G.nodes[node]["visited"] = True
        for edge in G.out_edges(node): # per ogni ramo del nodo visitato
            style_visited_edge((edge[0], edge[1], 0))
            neighbor = edge[1] # estraggo il nodo vicino
            weight = G.edges[(edge[0], edge[1], 0)]["weight"] # estraggo il peso del ramo
            # se la distanza del vicino è maggiore della distanza del nodo visitato + peso del ramo
            if G.nodes[neighbor]["distance"] > G.nodes[node]["distance"] + weight:
                G.nodes[neighbor]["distance"] = G.nodes[node]["distance"] + weight
                # salvi nel vicino il nodo migliore per raggiungerlo
                G.nodes[neighbor]["previous"] = node
                # aggiorniamo in pqueue il nuovo valore della distanza
                heapq.heappush(pq, (G.nodes[neighbor]["distance"], neighbor))
                for edge2 in G.out_edges(neighbor):
                    style_active_edge((edge2[0], edge2[1], 0))
        step += 1
```

La funzione che implementa l'algoritmo A* inizializza i nodi del grafo imposta il nodo di partenza, per poi utilizzare la coda di priorità per selezionare il nodo con il costo stimato più basso verso la destinazione, aggiornando iterativamente i punteggi g (costo reale) e f (costo stimato totale). I nodi visitati e i percorsi vengono tracciati per ricostruire la soluzione al termine.

```python
def a_star(orig, dest, plot=False):
    for node in G.nodes:
        G.nodes[node]["visited"] = False
        G.nodes[node]["distance"] = float("inf")
        G.nodes[node]["previous"] = None
        G.nodes[node]["size"] = 0
    for edge in G.edges:
        style_unvisited_edge(edge)
    G.nodes[orig]["distance"] = 0
    G.nodes[orig]["size"] = 50
    G.nodes[dest]["size"] = 50
    open_set = [(heuristicDistance(orig, dest), orig)]
    g_score = {node: float("inf") for node in G.nodes}
    g_score[orig] = 0
    f_score = {node: float("inf") for node in G.nodes}
    f_score[orig] = heuristicDistance(orig, dest)
    came_from = {}
    step = 0

    while open_set:
        _, current = heapq.heappop(open_set)

        if current == dest:
            # print("A* Iterations:", step)
            curr = dest
            while curr in came_from:
                G.nodes[curr]["previous"] = came_from[curr]
                curr = came_from[curr]
            return step

        if g_score[current] + heuristicDistance(current, dest) != f_score[current]:
            continue
        if G.nodes[current]["visited"]:
            continue
        G.nodes[current]["visited"] = True

        for u, v, key, data in G.out_edges(current, keys=True, data=True):
            neighbor = v
            weight = data.get("weight", float("inf"))
            tentative_g = g_score[current] + weight

            if tentative_g < g_score.get(neighbor, float("inf")):
                came_from[neighbor] = current
                g_score[neighbor] = tentative_g
                f_score[neighbor] = tentative_g + heuristicDistance(neighbor, dest)
                heapq.heappush(open_set, (f_score[neighbor], neighbor))
                style_visited_edge((u, v, key))
                for _, n, k in G.out_edges(neighbor, keys=True):
                    style_active_edge((neighbor, n, k))
        step += 1
```

4. Risultati e conclusioni

La comparazione tra Dijkstra e A* conferma le significative differenze teoriche anche sul piano pratico. Entrambi gli algoritmi condividono la stessa struttura di base per la ricerca del cammino minimo su grafi pesati, ma divergono per strategia di esplorazione ed efficienza computazionale.

La tabella e i grafici mostrano i risultati di un centinaio di elaborazioni sulle mappe di Amalfi, Cagliari e Roma, prese come esempi di grafi rispettivamente molto semplice, mediamente complesso e molto complesso.

Figura 1 - Amalfi

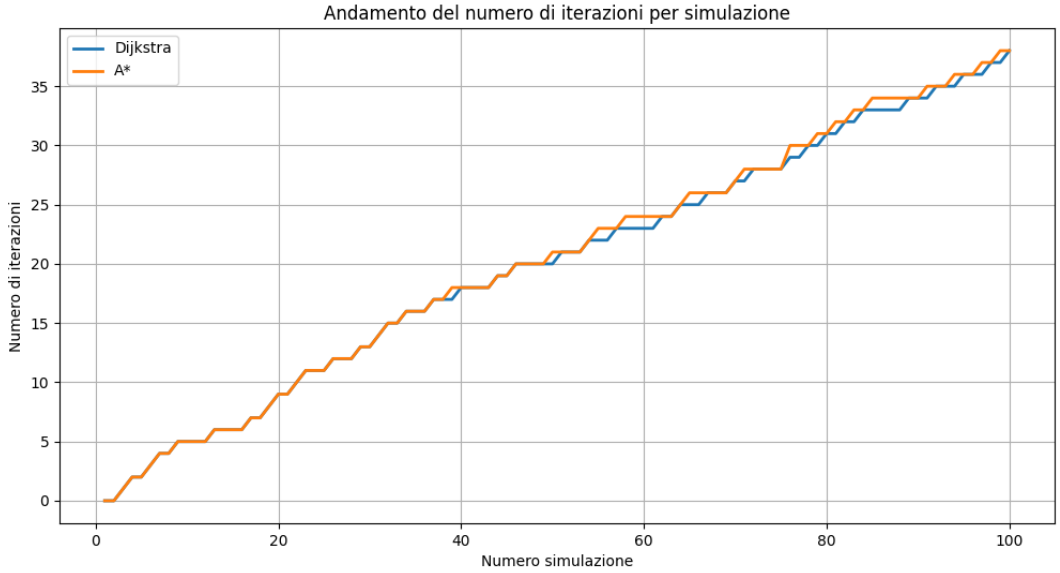

Figura 2 - Cagliari

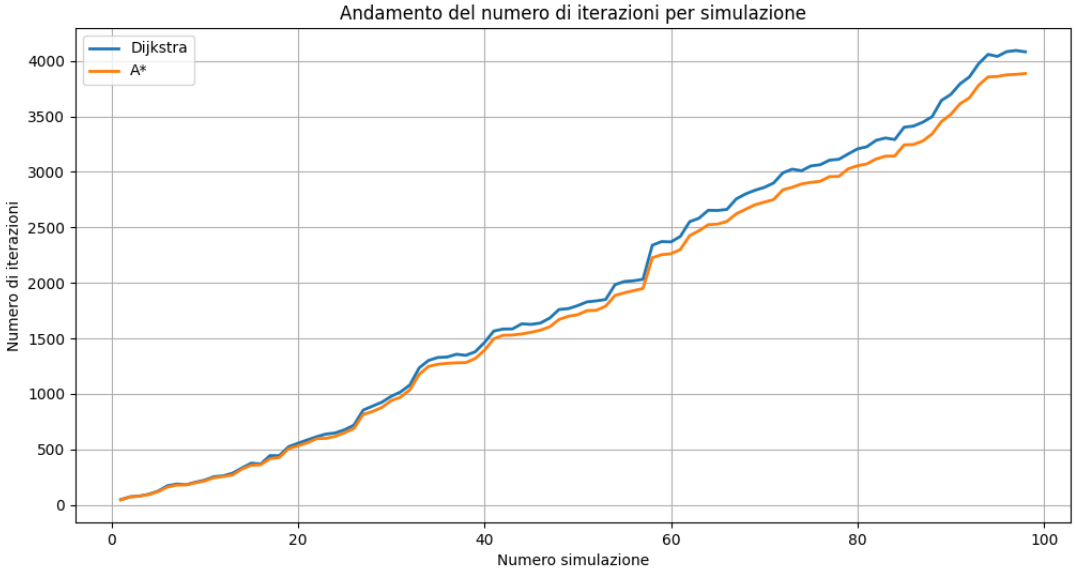

Figura 3 - Roma

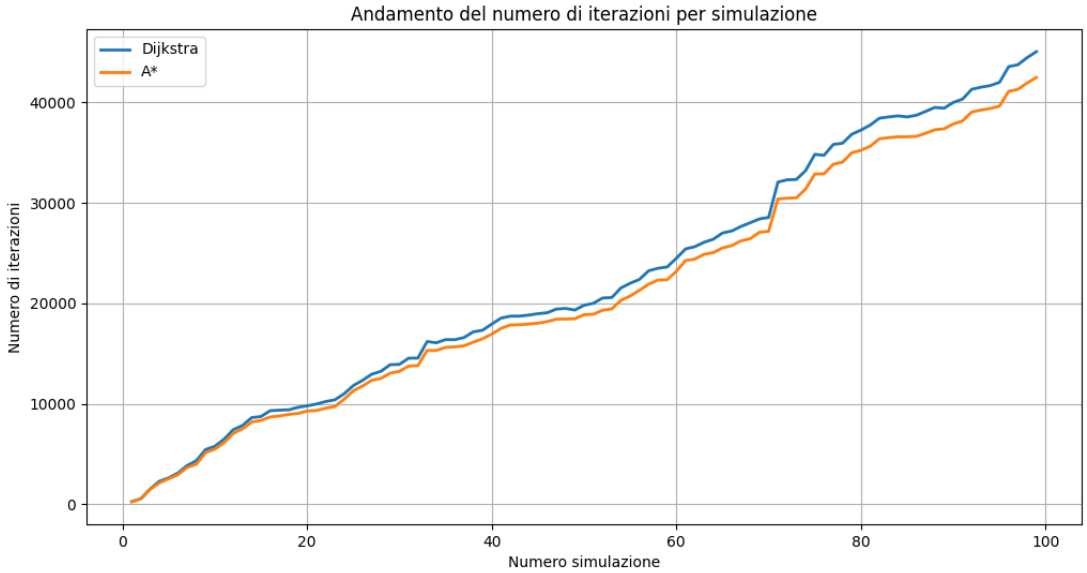

Tabella 1

Città	**Algoritmo**	**Iterazioni (min)**	**Iterazioni (max)**	**Iterazioni (mean)**	**Tempo (min)**	**Tempo (max)**	**Tempo (mean)**
Amalfi	Dijkstra	0	38	20.05	0.0001 s	0.0003 s	0.0002 s
	A*	0	38	20.29	0.0001 s	0.0004 s	0.0002 s
Cagliari	Dijkstra	45	4093	1902.63	0.0063 s	0.0462 s	0.0233 s
	A*	48	3886	1813.46	0.0071 s	0.0435 s	0.0225 s
Rome	Dijkstra	257	45062	22231.19	0.0825 s	0.5304 s	0.3087 s
	A*	271	42498	21046.34	0.0906 s	0.4980 s	0.2849 s

Si nota come l'efficacia di A*, paragonato a Dijkstra, cresce con la complessità della città. Infatti, nonostante le iterazioni del primo richiedano più tempo per la computazione, la sua efficienza permette un numero significativamente minore di iterazioni rispetto al secondo per raggiungere il percorso ottimale.

Abbiamo quindi conferma di come Dijkstra, esplorando indiscriminatamente tutti i percorsi possibili, risulti inefficiente in scenari complessi o di grandi dimensioni, laddove A* conserva l'ottimalità riducendo il numero di nodi visitati.

Per questo motivo, A* è preferibile per applicazioni di pathfinding in cui si dispone di informazioni sulla posizione della destinazione e in contesti con grafi molto estesi.

Possibili miglioramenti e sviluppi futuri possono includere:

- Euristiche personalizzate: progettare euristiche su misura, ad esempio basate sul profilo del traffico, sulla priorità delle strade o su vincoli ambientali (ZTL, pendenze), può rendere l'algoritmo ancora più performante e adatto a casi d'uso reali.
- Integrazione con dati dinamici: combinare A* con dati in tempo reale, come condizioni del traffico o lavori stradali, consente un pathfinding adattivo, più vicino al comportamento di un sistema di navigazione moderno.

In conclusione, la scelta tra Dijkstra e A* non dipende solo dal tipo di grafo, ma anche dal contesto applicativo e dalle risorse disponibili. A* rappresenta generalmente la scelta preferita per i veicoli autonomi grazie al miglior compromesso tra precisione e velocità, soprattutto se accompagnato da un'euristica efficace.