RICERCA_DI_SOLUZIONI_IN_SPAZI_DI_STATI

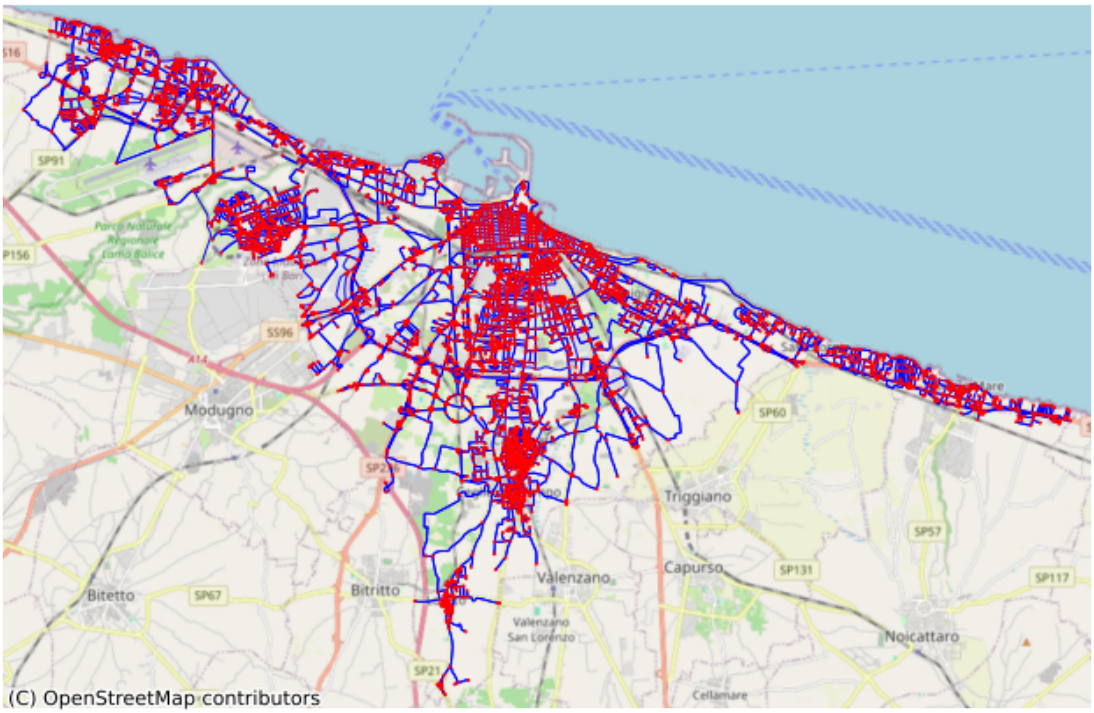
February 7, 2025

0.1 Introduzione

Per la trattazione di questo argomento, l'obiettivo fissato è stato quello testare diversi algoritmi di ricerca per determinare il percorso più sicuro in una rete stradale urbana, nello specifico della città di Bari. Di fatto, ogni algoritmo è stato testato sulla ricerca di un percorso da Piazza Umberto Primo a Parco Maria Maugeri.

0.2 Rappresentazione della conoscenza per KB

In questo caso, la conoscenza è rappresentata utilizzando un grafo **G** che mappa la **rete stradale della città di Bari**. Ogni nodo del grafo rappresenta un incrocio o un punto di interesse, mentre gli archi collegano questi nodi e rappresentano i segmenti stradali tra i vari punti. Inoltre, ogni arco è stato arricchito con le informazioni relative al **numero di incidenti** avvenuti su quel segmento stradale, tratte da un dataset previamente elaborato grazie a dati forniti da enti regionali che si occupano dell'acquisizione e della rappresentazione di tali informazioni. Di seguito un'immagine illustrativa del grafo rappresentativo della rete stradale:



0.2.1 Librerie utilizzate

[3]:
```
import osmnx as ox  # lutile per scaricare, modellare, analizzare e␣
 ↪visualizzare facilmente reti stradali e altre funzionalità geospaziali da␣
 ↪OpenStreetMap
import folium       # lutile per creare mappe interattive
import networkx as nx  #lutile per la creazione e manipolazione di grafi␣
 ↪complessi.
import matplotlib.pyplot as plt  #lutile per la creazione di grafici e␣
 ↪visualizzazioni
import pandas as pd  #lutile per la manipolazione e l'analisi dei dati
import heapq #lutile per la gestione di code di priorità
import time #lutile per la gestione del tempo
import sys  #lutile per la gestione del sistema
from geopy.distance import geodesic #lutile per calcolare la distanza tra due␣
 ↪punti sulla Terra
import contextily as ctx  #lutile per visualizzare mappe di base di sfondo
```

Estrazione della rete stradale di Bari e salvataggio nel grafo G

[ ]:
```
#Estrazione della rete stradale di Bari
G = ox.graph_from_place("Bari, Italy", network_type="drive", simplify=True)

#Salvare la rete in formato GraphML
ox.save_graphml(G, "rete_bari.graphml")
```

Plot del grafo

[ ]:
```
# Carica il grafo
G = ox.load_graphml("rete_bari.graphml")

# Plotta il grafo senza chiudere la figura
fig, ax = ox.plot_graph(G, node_size=2, edge_linewidth=0.7, edge_color="blue",␣
 ↪node_color="red", show=False, close=False)

# Aggiunta della mappa come sfondo
ctx.add_basemap(ax, source=ctx.providers.OpenStreetMap.Mapnik, crs=G.
 ↪graph['crs'])

plt.show()
```

0.3 Aggiunta numero incidenti sui segmenti stradali

Sfruttando il dataset **"Dati_aggregati"**, dove per ogni segmento stradale sono presenti il *numero di incidenti* avvenuti, aggiorno gli archi del grafo G con questa informazione

[ ]:
```
df = pd.read_csv(".\\Dati\\Dati_aggregati.csv")

incidenti_dict = df.groupby("ID_Segmento")["NUM_INCIDENTI"].sum().to_dict()
```

```
for u, v, k, data in G.edges(data=True, keys=True):
    if "osmid" in data:
        osmids = data["osmid"]

        if isinstance(osmids, list):
            incidenti_count = sum(incidenti_dict.get(osmid, 0) for osmid in␣
 ↪osmids)
        else:
            incidenti_count = incidenti_dict.get(osmids, 0)

        data["incidenti"] = incidenti_count

ox.save_graphml(G, "rete_bari_incidenti.graphml")
print("Grafo salvato con successo")
```

Script utile a trovare il nodo più vicino a delle coordinate inserite

[ ]:
```
def get_nearest_node(graph, x, y):
    # Trovare il nodo più vicino al punto di input
    nearest_node = ox.distance.nearest_nodes(graph, X=x, Y=y)
    nodes, _ = ox.graph_to_gdfs(graph)
    node_coords = nodes.loc[nearest_node].geometry

    return nearest_node, node_coords.y, node_coords.x

G = ox.load_graphml("rete_bari_incidenti.graphml")

x_input = 16.8548679  # Longitudine
y_input = 41.1251951  # Latitudine

node_id, lat, lon = get_nearest_node(G, x_input, y_input)

print(f"Nodo più vicino trovato: ID = {node_id}, Latitudine = {lat},␣
 ↪Longitudine = {lon}")
```

1 RICERCA NON INFORMATA

1.1 ITERATIVE DEEPENING DEPHT FIRST SEARCH

Per la ricerca non informata è stato l'implementato l'***Iterative Deepening Depth First Search***. L'algoritmo **Iterative Deepening Depth First Search (IDDFS)** è un ibrido che combina le caratteristiche della **ricerca in profondità (DFS)** e della **ricerca per livello (BFS)**. La sua peculiarità risiede nel fatto che esegue ripetutamente una ricerca in profondità, ma limitando la profondità massima per ogni iterazione. Ad ogni ciclo, la profondità massima viene incrementata, fino a trovare il percorso desiderato o esaurire lo spazio di ricerca. Nel codice proposto, l'algoritmo è composto da due funzioni principali:

1. **Funzione `dls` (Depth-Limited Search):** Questa funzione esegue una ricerca in profondità

limitata, in cui si esplorano i nodi del grafo a partire dal nodo iniziale (start), fino a una profondità massima specificata. Ogni volta che si esplora un nodo, se non è già stato visitato, viene aggiunto allo stack con il percorso accumulato fino a quel punto. Se il nodo di arrivo (goal) viene raggiunto, la funzione restituisce il percorso; altrimenti, la ricerca continua fino a raggiungere la profondità limite.

2. **Funzione `iterative_deepening_dfs`:** Questa funzione è responsabile dell'iterazione del processo di ricerca in profondità. Ad ogni iterazione, l'algoritmo incrementa la profondità massima da 0 fino al valore massimo di **`max_depth`**. Ogni volta che la ricerca raggiunge la profondità massima senza trovare la soluzione, viene incrementata la profondità e la ricerca viene ripetuta. In caso di successo, l'algoritmo restituisce il **percorso trovato** insieme ad alcune statistiche, come la **profondità raggiunta**, il **tempo di esecuzione** e la **memoria utilizzata**

[5]:
```
"""Ricerca in profondità limitata"""
def dls(graph, start, goal, depth, explored_paths):
    stack = [(start, [start])]

    while stack:
        node, path = stack.pop()
        explored_paths.append(path)  # Salva il percorso esplorato

        if node == goal:
            return path  # Ritorna il percorso trovato

        if len(path) - 1 < depth:  # Controllo della profondità
            for neighbor in graph.neighbors(node):
                if neighbor not in path:
                    stack.append((neighbor, path + [neighbor]))

    return None  # Nessun percorso trovato entro la profondità

def iterative_deepening_dfs(graph, start, goal, max_depth=22):
    explored_paths = [] # Memorizza tutti i percorsi esplorati
    start_time = time.time() # Inizia a calcolare il tempo di esecuzione
    memory_before = sys.getsizeof(explored_paths) # Calcola la memoria␣
 ↪utilizzata prima dell'esecuzione

    for depth in range(max_depth):
        result = dls(graph, start, goal, depth, explored_paths)
        if result:
            execution_time = time.time() - start_time
            memory_after = sys.getsizeof(explored_paths)
            memory_used = memory_after - memory_before
            return result, explored_paths, depth, execution_time, memory_used ␣
 ↪# Restituisce il percorso, profondità, tempo e memoria

    execution_time = time.time() - start_time
```

```
    memory_after = sys.getsizeof(explored_paths)
    memory_used = memory_after - memory_before
    return None, explored_paths, max_depth, execution_time, memory_used  #␣
 ↪Nessun percorso trovato

# Caricare il grafo di Bari
G = ox.load_graphml("Dati\\rete_bari_incidenti.graphml")

partenza = 270659688 # ID del nodo di partenza
arrivo = 1481415203  # ID del nodo di arrivo

# Trovare il percorso più breve tra i due nodi e raccogliere statistiche
shortest_path, explored_paths, depth_reached, exec_time, memory_used =␣
 ↪iterative_deepening_dfs(G, partenza, arrivo)

print("-- RICERCA CON IDDFS COMPLETATA --")

if shortest_path:
    print(f"\nPercorso più breve trovato: {shortest_path}")
    print(f"Profondità raggiunta: {depth_reached}")
    print(f"Percorsi esplorati: {len(explored_paths)}")
    print(f"Tempo di esecuzione: {exec_time:.4f} secondi")
    print(f"Memoria utilizzata: {memory_used} byte")
else:
    print("\nNessun percorso trovato.")
    print(f"Profondità raggiunta: {depth_reached}")
    print(f"Tempo di esecuzione: {exec_time:.4f} secondi")
    print(f"Memoria utilizzata: {memory_used} byte")
```

```
-- RICERCA CON IDDFS COMPLETATA --

Percorso più breve trovato: [270659688, 270388628, 322548994, 322549051,
322550079, 322549607, 322550392, 322550761, 320970977, 330655154, 330655155,
270388358, 320971935, 320971934, 270437927, 353330854, 270389790, 329988604,
329988602, 270655174, 270654641, 1481415203]
Profondità raggiunta: 21
Percorsi esplorati: 63991
Tempo di esecuzione: 0.0755 secondi
Memoria utilizzata: 562432 byte
```

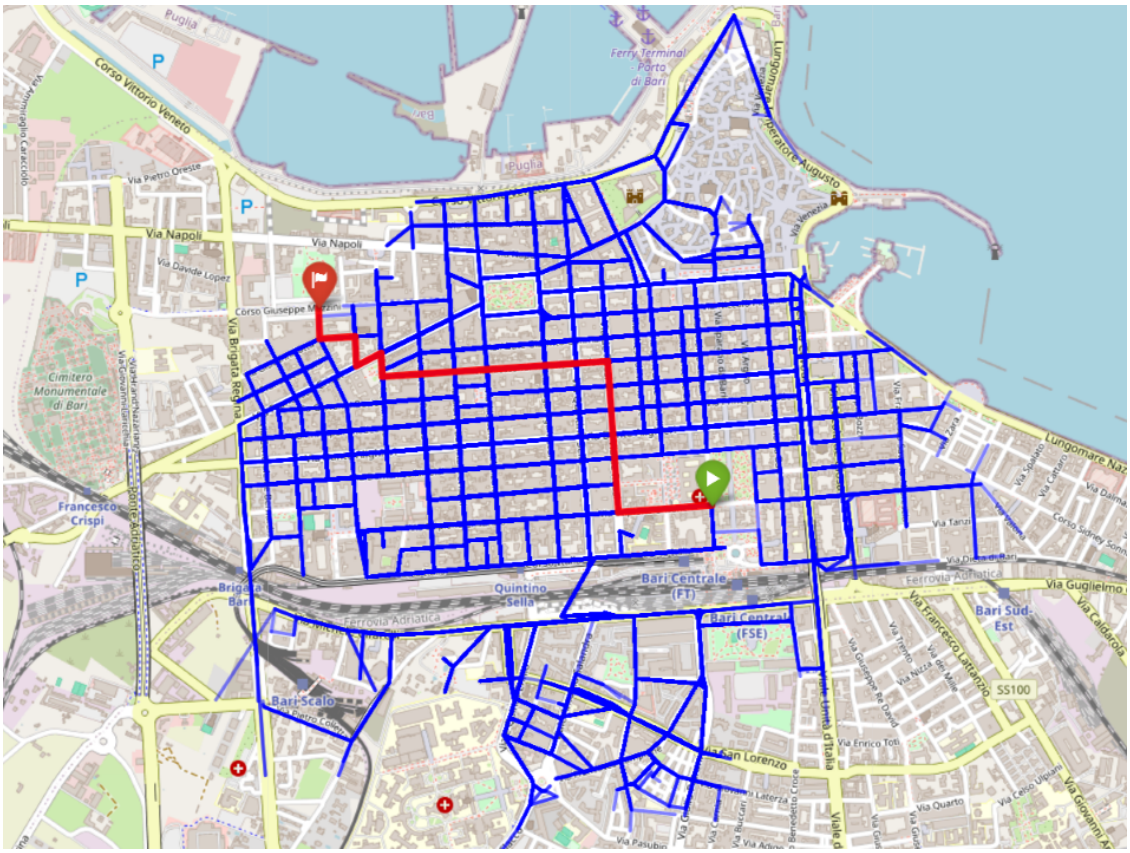
1.1.1 Valutazione

Dopo aver eseguito l'algoritmo settando il parametro **max_depth = 50** (per permettere all'algoritmo di esplorare una quantità significativa di soluzioni prima di terminare, senza limitare eccessivamente la profondità di ricerca) i risultati ottenuti sono i seguenti:

- **Percorso più breve trovato (in rosso sulla mappa):** [270659688, 270388628, 322548994, 322549051, 322550079, 322549607, 322550392, 322550761, 320970977, 330655154, 330655155,

270388358, 320971935, 320971934, 270437927, 353330854, 270389790, 329988604, 329988602, 270655174, 270654641, 1481415203].
Ogni elemento all'interno della lista rappresenta l'id del nodo attraversato per arrivare alla soluzione

- **Percorsi totali esplorati (in blu sulla mappa):** *63991*
- **Pronfondità raggiunta:** *21*, ciò significa che per navigare dal punto di partenza a quello di arrivo sono necessari bisogna attraversare 21 segmenti (archi) nella rete.
- **Tempo di esecuzione:** *0.0755 secondi*
- **Memoria utilizzata:** *562432 byte*



1.1.2 APPLICAZIONE MULTIPLE-PATH PRUNING

Per ottimizzare IDDFS è stata applicata la tecnica del **Multiple-Path Pruning** per evitare la riesplorazione di percorsi già esplorati. Questa tecnica comporta un miglioramento nell'efficienza della ricerca, in quanto impedisce di esplorare ripetutamente percorsi che non conducono a soluzioni migliori o più rapide.
Nella **versione precedente**, l'algoritmo eseguiva una ricerca in profondità limitata (DLS) senza tenere traccia in modo efficiente dei percorsi già esplorati. Ciò significava che, in alcuni casi, venivano riesplorati gli stessi percorsi, aumentando il numero di nodi esplorati e il consumo di memoria.
Con la versione ottimizzata:

1. **Pruning dei percorsi ridondanti:** L'algoritmo ora utilizza un dizionario `visited`, per evitare di esplorare nodi già visitati con un percorso più lungo. Questo riduce notevolmente il numero di percorsi esplorati.

2. **Ottimizzazione del consumo di memoria:** Poiché vengono evitati percorsi ripetitivi, la memoria utilizzata è significativamente ridotta rispetto alla versione precedente.

3. **Maggiore efficienza:** La ricerca avviene più velocemente, con un numero minore di percorsi esplorati, migliorando il tempo di esecuzione complessivo.

[7]:
```python
"""Ricerca in profondità con MPP"""
def dls(graph, start, goal, depth, explored_paths, visited):
    stack = [(start, [start])]

    while stack:
        node, path = stack.pop()
        # evita di esplorare nodi già visitati con un percorso più lungo
        if node in visited and len(path) - 1 > visited[node]:
            continue

        visited[node] = len(path) - 1  # Aggiornamento livello minimo raggiunto␣
↪per questo nodo
        explored_paths.append(path)  # Salvataggio percorso esplorato

        if node == goal:
            return path  # Ritorna il percorso trovato

        if len(path) - 1 < depth:  # Controllo della profondità
            for neighbor in graph.neighbors(node):
                if neighbor not in path:
                    stack.append((neighbor, path + [neighbor]))

    return None  # Nessun percorso trovato entro la profondità

def iterative_deepening_dfs(graph, start, goal, max_depth=50):
    explored_paths = []
    start_time = time.time()
    memory_before = sys.getsizeof(explored_paths)

    visited = {}  # Dizionario per tracciare la profondità minima raggiunta per␣
↪ogni nodo

    for depth in range(max_depth):
        result = dls(graph, start, goal, depth, explored_paths, visited)
        if result:
            execution_time = time.time() - start_time
```

```
            memory_after = sys.getsizeof(explored_paths)
            memory_used = memory_after - memory_before
            return result, explored_paths, depth, execution_time, memory_used ␣
 ↪# Restituisce il percorso, profondità, tempo e memoria

    execution_time = time.time() - start_time
    memory_after = sys.getsizeof(explored_paths)
    memory_used = memory_after - memory_before
    return None, explored_paths, max_depth, execution_time, memory_used  #␣
 ↪Nessun percorso trovato

# Caricare il grafo di Bari
G = ox.load_graphml("Dati\\rete_bari_incidenti.graphml")

partenza = 270659688 # ID del nodo di partenza
arrivo = 1481415203  # ID del nodo di arrivo

# Trovare il percorso più breve tra i due nodi e raccogliere statistiche
shortest_path, explored_paths, depth_reached, exec_time, memory_used =␣
 ↪iterative_deepening_dfs(G, partenza, arrivo)

print("-- RICERCA CON IDDFS OTTIMIZZATA COMPLETATA --")

if shortest_path:
    print(f"\nPercorso più breve trovato: {shortest_path}")
    print(f"Profondità raggiunta: {depth_reached}")
    print(f"Percorsi esplorati: {len(explored_paths)}")
    print(f"Tempo di esecuzione: {exec_time:.4f} secondi")
    print(f"Memoria utilizzata: {memory_used} byte")
else:
    print("Nessun percorso trovato.")
    print(f"Profondità raggiunta: {depth_reached}")
    print(f"Tempo di esecuzione: {exec_time:.4f} secondi")
    print(f"Memoria utilizzata: {memory_used} byte")
```

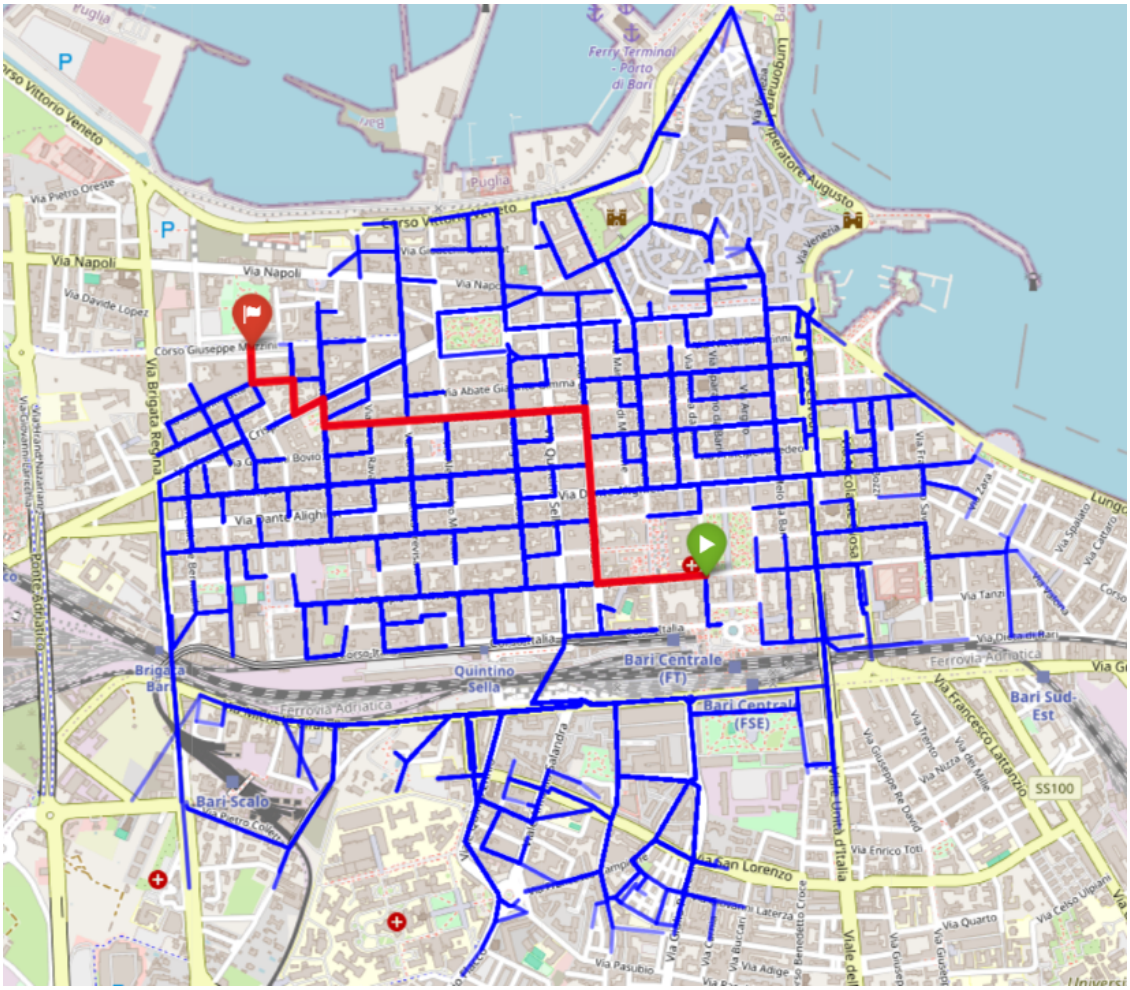
```
-- RICERCA CON IDDFS OTTIMIZZATA COMPLETATA --

Percorso più breve trovato: [270659688, 270388628, 322548994, 322549051,
322550079, 322549607, 322550392, 322550761, 320970977, 330655154, 330655155,
270388358, 320971935, 320971934, 270437927, 353330854, 270389790, 329988604,
329988602, 270655174, 270654641, 1481415203]
Profondità raggiunta: 21
Percorsi esplorati: 8020
Tempo di esecuzione: 0.0156 secondi
Memoria utilizzata: 67168 byte
```

1.1.3 Valutazione

L'esecuzione dell'algoritmo ottimizzato ha riportato i risultati ottenuti sono i seguenti:

- **Percorso più breve trovato (in rosso sulla mappa):** [270659688, 270388628, 322548994, 322549051, 322550079, 322549607, 322550392, 322550761, 320970977, 330655154, 330655155, 270388358, 320971935, 320971934, 270437927, 353330854, 270389790, 329988604, 329988602, 270655174, 270654641, 1481415203].
  Uguale a quello trovato precedentemente
- **Percorsi totali esplorati (in blu sulla mappa):** *8020* Circa l'*87% in meno* rispetto all'implementazione senza MPP.
- **Pronfondità raggiunta:** *21*,
- **Tempo di esecuzione:** *0.0126 secondi*
- **Memoria utilizzata:** *67168 byte*

Dalla seuguente immagine possiamo notare che la rete dei percorsi esplorati è meno fitta rispetto a quella esplorata dall'algoritmo senza l'utilizzo di MPP



1.1.4 Confronti e conclusioni

RISULTATO	IDDFS	IDDFS CON MPP
PERCORSO TROVATO	Stesso	Stesso
PROFONDITÀ RAGGIUNTA	21	21
PERCORSI ESPLORATI	63991	8020
TEMPO DI ESECUZIONE	0.0755 s	0.0156 s
MEMORIA UTILIZZATA	562432 byte	67168 byte

L'algoritmo ottimizzato ha portato a **un numero significativamente inferiore di percorsi esplorati** (da 63991 a 8020), a un **miglioramento notevole nel tempo di esecuzione** (da 0.0755 a 0.0156 secondi) e a una **riduzione drastica della memoria utilizzata** (da 562432 byte a 67168 byte). Questi miglioramenti sono il risultato diretto dell'implementazione del **Multiple-Path Pruning**, che ha permesso di evitare la riesplorazione dei percorsi già visitati, ottimizzando l'efficienza complessiva dell'algoritmo.

Funzione utile a plottare su mappa i percorsi esplorati

[ ]:
```
def plot_paths_on_map(graph, shortest_path, explored_paths, start, goal):
    """Visualizza tutti i percorsi esplorati e il percorso più breve sulla␣
↪mappa."""
    nodes = ox.graph_to_gdfs(graph, nodes=True, edges=False)
    center = (nodes.loc[start].geometry.y, nodes.loc[start].geometry.x)

    mappa = folium.Map(location=center, zoom_start=13)

    # Disegnare tutti i percorsi esplorati in blu
    for path in explored_paths:
        path_coords = [(nodes.loc[node].geometry.y, nodes.loc[node].geometry.x)␣
↪for node in path]
        folium.PolyLine(path_coords, color="blue", weight=3, opacity=0.5).
↪add_to(mappa)

    # Disegnare il percorso più breve in rosso sopra il blu
    if shortest_path:
        path_coords = [(nodes.loc[node].geometry.y, nodes.loc[node].geometry.x)␣
↪for node in shortest_path]
        folium.PolyLine(path_coords, color="red", weight=6, opacity=0.9).
↪add_to(mappa)

    # Aggiungere marker per il punto di partenza e di arrivo
    folium.Marker(
        location=(nodes.loc[start].geometry.y, nodes.loc[start].geometry.x),
        popup="Partenza",
        icon=folium.Icon(color="green", icon="play")
    ).add_to(mappa)

    folium.Marker(
        location=(nodes.loc[goal].geometry.y, nodes.loc[goal].geometry.x),
```

```
        popup="Arrivo",
        icon=folium.Icon(color="red", icon="flag")
    ).add_to(mappa)

    return mappa

mappa = plot_paths_on_map(G, shortest_path, explored_paths, partenza, arrivo)
mappa.save("percorso_IDDFS.html")
```

2 RICERCA A COSTO MINIMO

2.1 LOWEST COST-FIRST SEARCH

Per questo argomento è stato implementato l'algoritmo **Lowest-Cost-First Search (LCFS)**, una strategia che seleziona il percorso con il costo cumulativo minimo. Nel contesto specifico, il "costo" è stato definito come il numero di **incidenti** registrati sugli archi stradali, rendendo **l'obiettivo** quello di ***identificare il percorso con il minor rischio di incidentalità.***

2.1.1 Multiple Path Pruning

L'algoritmo implementa un'ottimizzazione attraverso la tecnica del **multiple path pruning**.
Il dizionario `visited` memorizza il costo minimo storico per raggiungere ciascun nodo. Quando un nodo viene estratto dalla coda:

1. Si confronta il costo del percorso corrente con quello memorizzato in `visited`
2. Se il costo corrente è maggiore o uguale al costo memorizzato, il percorso viene **scartato**
3. Solo i percorsi con costo inferiore a quello registrato vengono **processati ed espansi**

Benefici della tecnica:
- **Eliminazione di percorsi ridondanti**: evita di riesplorare nodi attraverso cammini più costosi
- **Ottimizzazione delle risorse**: riduce lo spazio di ricerca e il tempo di esecuzione
- **Convergenza garantita**: previene loop infiniti e cicli non ottimali

2.1.2 Struttura dell'Algoritmo

L'algoritmo utilizza una **coda a priorità** (implementata tramite `heapq`) per gestire i nodi da esplorare, **ordinandoli in base al costo cumulativo**.
Per ogni nodo, vengono mantenuti:

- **Costo cumulativo**: somma degli incidenti lungo il percorso
- **Nodo corrente**
- **Percorso accumulato**

Il **grafo** è stato caricato con **attributi personalizzati (incidenti) sugli archi**, rappresentanti il numero di incidenti storici.
L'algoritmo evita **cicli e percorsi ridondanti** memorizzando il costo minimo per raggiungere ciascun nodo in un dizionario `visited`.

[ ]:
```
"""Lowest-Cost-First Search"""
def lowest_cost_first_search(graph, start, goal):

    priority_queue = [(0, start, [start])]  # (costo cumulativo, nodo attuale, percorso)
    visited = {}  # Dizionario per memorizzare il miglior costo trovato per ciascun nodo
    explored_paths = []  # Lista dei percorsi esplorati

    start_time = time.time()
    memory_before = sys.getsizeof(priority_queue) + sys.getsizeof(visited) + sys.getsizeof(explored_paths)
    max_depth = 0

    while priority_queue: # Finchè la coda di priorità non è vuota
        cost, node, path = heapq.heappop(priority_queue)  # Estrazione del nodo con il minor costo
        explored_paths.append((path, cost))  # Memorizza il percorso esplorato
        max_depth = max(max_depth, len(path) - 1) # Aggiorna la profondità massima raggiunta

        if node == goal:
            execution_time = time.time() - start_time
            memory_after = sys.getsizeof(priority_queue) + sys.getsizeof(visited) + sys.getsizeof(explored_paths)
            memory_used = memory_after - memory_before
            return explored_paths, path, cost, len(explored_paths), execution_time, memory_used, max_depth # Percorso trovato

        if node in visited and cost >= visited[node]:
            continue  # Se il nodo è già stato visitato con un costo inferiore, lo ignoriamo

        visited[node] = cost  # Memorizza il costo minimo per raggiungere questo nodo

        for neighbor in graph.neighbors(node):
            edge_data = graph.get_edge_data(node, neighbor)

            # Gestione di archi multipli tra due nodi
            if isinstance(edge_data, dict):
                edge_data = min(edge_data.values(), key=lambda d: d.get("incidenti", float('inf'))) # Trova l'arco con il minor numero di incidenti se ci sono più archi tra due nodi
```

```
            edge_cost = int(edge_data.get("incidenti", 0))  # Converte il␣
 ↪valore in intero, 0 se null
            new_cost = cost + edge_cost
            heapq.heappush(priority_queue, (new_cost, neighbor, path +␣
 ↪[neighbor]))

    execution_time = time.time() - start_time
    memory_after = sys.getsizeof(priority_queue) + sys.getsizeof(visited) + sys.
 ↪getsizeof(explored_paths)
    memory_used = memory_after - memory_before
    return explored_paths, None, float('inf'), len(explored_paths),␣
 ↪execution_time, memory_used, max_depth

# Caricare il grafo con il numero di incidenti sugli archi
G = ox.load_graphml("..\\Dati\\rete_bari_incidenti.graphml")

partenza = 270659688
arrivo = 1481415203

# Eseguire la ricerca Lowest-Cost-First basata sugli incidenti
explored_paths, shortest_path, min_incidents, num_explored, exec_time,␣
 ↪memory_used, max_depth = lowest_cost_first_search(G, partenza, arrivo)

print(f"Numero totale di percorsi esplorati: {num_explored}")
print(f"Tempo di esecuzione: {exec_time:.4f} secondi")
print(f"Memoria utilizzata: {memory_used} byte")
print(f"Profondità massima raggiunta: {max_depth}")

if shortest_path:
    print(f"\nPercorso con il minor numero di incidenti trovato:␣
 ↪{shortest_path}")
    print(f"Numero totale di incidenti lungo il percorso: {min_incidents}")
else:
    print("Nessun percorso trovato.")
```

```
Numero totale di percorsi esplorati: 6239
Tempo di esecuzione: 0.0329 secondi
Memoria utilizzata: 205232 byte
Profondità massima raggiunta: 129

Percorso con il minor numero di incidenti trovato: [270659688, 270388628,
322548994, 322548996, 12083787147, 569212252, 3860917204, 3841272949,
3860940020, 330076413, 279650482, 1483634369, 1483634284, 279655605, 280884748,
279380962, 339607009, 279655525, 10680697917, 316572227, 10680697906, 279655521,
9602691583, 1014703348, 1249973250, 4395827572, 5395447618, 5395447592,
5395448236, 5395447590, 330932109, 1459966804, 298502133, 329994613, 330003439,
1481415203]
```

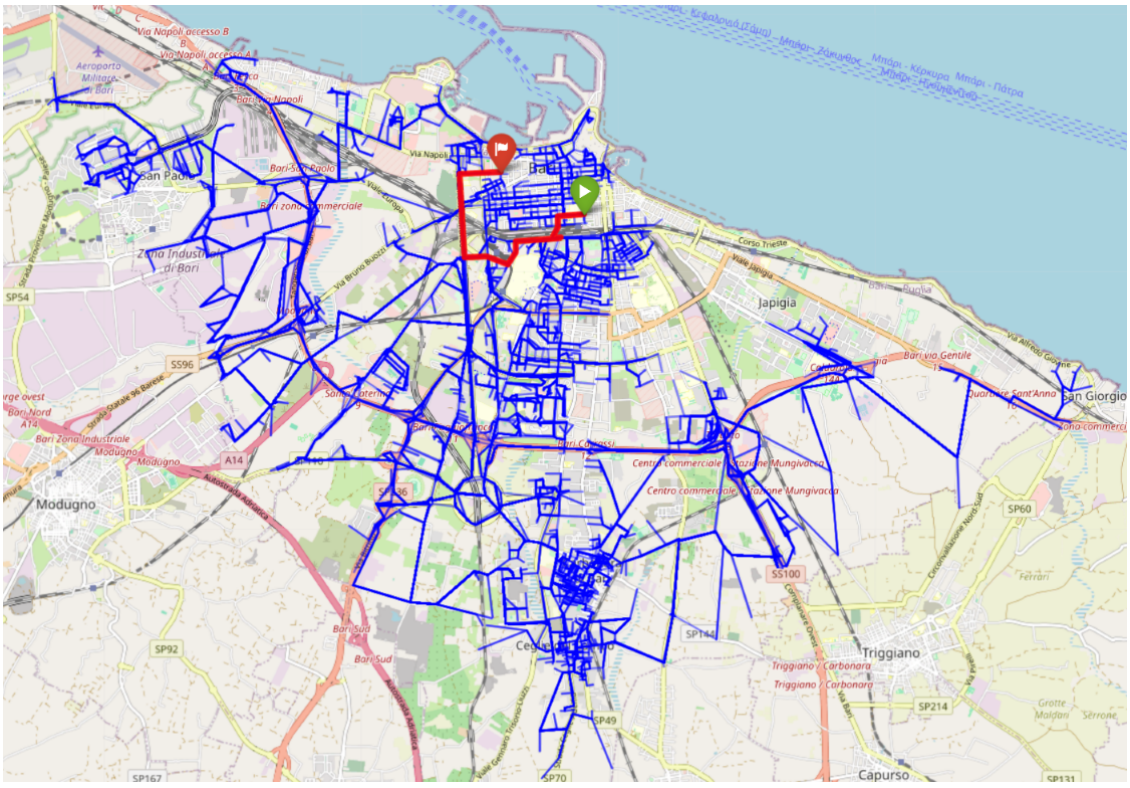
```
Numero totale di incidenti lungo il percorso: 123
```

2.1.3 Valutazione

L'esecuzione dell'algoritmo ha riportato i risultati ottenuti sono i seguenti:

- **Percorso più breve trovato (in rosso sulla mappa):** [270659688, 270388628, 322548994, 322548996, 12083787147, 569212252, 3860917204, 3841272949, 3860940020, 330076413, 279650482, 1483634369, 1483634284, 279655605, 280884748, 279380962, 339607009, 279655525, 10680697917, 316572227, 10680697906, 279655521, 9602691583, 1014703348, 1249973250, 4395827572, 5395447618, 5395447592, 5395448236, 5395447590, 330932109, 1459966804, 298502133, 329994613, 330003439, 1481415203]
- **Percorsi totali esplorati (in blu sulla mappa):** *6239*
- **Numero totale di incidenti lungo il percorso:** *123*
- **Pronfondità raggiunta:** *129*,
- **Tempo di esecuzione:** *0.0329 secondi*
- **Memoria utilizzata:** *205232 byte*

OSSERVAZIONE: *Dalla seguente immagine possiamo notare che il percorso con il minor numero di incidenti prende in considerazione una strada exstraurbana, sulla quale possiamo immaginare che avvengano meno incidenti rispetto all'area urbana*



3 RICERCA INFORMATA

La ricerca informata utilizza conoscenza specifica del dominio per guidare l'esplorazione dello spazio degli stati in modo più efficiente rispetto alle strategie non informate. A differenza di algoritmi come IDDFS e LcFS, sfrutta **euristiche** per stimare il costo residuo verso il goal, privilegiando i percorsi più promettenti. Questo riduce drasticamente il numero di nodi esplorati, migliorando tempi e risorse.

3.0.1 EURISTICHE UTILIZZATE

Euristica 1:

$$h_1(n) = D_g(n, \text{goal}) \times \left(\frac{\text{incidenti\_medi\_per\_km}}{\text{lunghezza\_media\_archi}} \right)$$

con:
- $D_g(n, \text{goal})$: Distanza geodetica in metri tra il nodo (n) e il goal.
- incidenti_medi_per_km: Numero medio di incidenti per chilometro calcolato su tutto il grafo.
- lunghezza_media_archi: Lunghezza media degli archi nel grafo (in metri).

Problemi di Ammissibilità e Consistenza dell'Euristica 1

L'euristica $h_1(n)$, pur cercando di combinare distanza e rischio di incidenti, presenta gravi problemi che ne compromettono l'affidabilità, tanto da renderla inadatta in applicazioni pratiche. Ecco un riassunto delle principali problematiche:

1. **Inammissibilità**: L'errore principale è il **problema dimensionale**: l'euristica combina due grandezze incompatibili (incidenti per chilometro e lunghezza media degli archi in metri), creando un'unità di misura senza senso fisico. Questo porta a una **sovrastima** sistematica del costo atteso, violando la condizione di **ammissibilità** (l'euristica non deve mai superare il costo reale del percorso).

2. **Inconsistenza**: La sovrastima causata dall'errore dimensionale impedisce il rispetto della **disuguaglianza triangolare**, che è un requisito essenziale per la **consistenza**. Questo potrebbe portare l'algoritmo A* a ignorare percorsi validi, fallendo nel trovare soluzioni ottimali o addirittura non trovando alcuna soluzione.

3. **Implicazioni Pratiche**: L'euristica $h_1(n)$ non è adatta per contesti in cui è fondamentale ottenere soluzioni ottimali. Sebbene possa essere utilizzata in scenari esplorativi per velocizzare l'esecuzione, in applicazioni reali, come la navigazione urbana, l'uso di questa euristica potrebbe generare percorsi più rischiosi.

Conclusione
L'euristica $h_1(n)$, nonostante sembri intuitiva, fallisce nel garantire le proprietà teoriche necessarie per un'applicazione corretta, rendendola inadatta per garantire l'ottimalità e la sicurezza nei percorsi. Il suo utilizzo dovrebbe essere limitato a contesti dove si accettano compromessi sulla sicurezza a favore di una maggiore velocità computazionale.

Euristica 2: L'euristica $h_2(n)$ si basa sulla **distanza geodetica** tra il nodo corrente e l'obiettivo, ponderata con il **minimo tasso di incidenti per chilometro**. La formula dell'euristica è:

$$h_2(n) = \text{distanza}(n, \text{goal}) \times \text{min\_rate}$$

dove:

- distanza(n, goal) è la distanza geodetica tra il nodo (n) e l'obiettivo (in chilometri),
- min_rate è il tasso minimo di incidenti per chilometro nel grafo.

L'**euristica** $h_2(n)$ stima il costo di un percorso tra un nodo e l'obiettivo moltiplicando la **distanza geodetica** tra i due nodi per un fattore che rappresenta il **minimo tasso di incidenti per chilometro** nel grafo. La funzione `min_incident_rate_per_km` calcola il tasso minimo di incidenti per chilometro lungo tutti i segmenti stradali nel grafo.

Ammissibilità: Questa euristica è **ammissibile** poiché, per definizione, **non sovrastima mai il costo reale** del percorso. Il tasso minimo di incidenti per chilometro è una stima inferiore o uguale alla densità di incidenti che si riscontra effettivamente lungo il percorso. Pertanto, la stima non può mai essere maggiore del costo reale del percorso.

Consistenza: L'euristica è **consistente** in quanto non violerà mai la disuguaglianza triangolare. Poiché si basa sulla distanza geodetica e sul tasso minimo di incidenti, la somma delle stime delle euristiche sui nodi intermedi non sarà mai maggiore della stima di un percorso diretto.

In sintesi, **l'euristica** $h_2(n)$ è un'euristica ammissibile e consistente, fornendo una buona stima del costo del percorso in base alla distanza e al rischio di incidenti lungo il percorso.

[8]:
```
"""EURISTICHE PER LA RICERCA INFORMATA"""

def geodetic_distance(graph, node1, node2):
    """Restituisce la distanza geodetica tra due nodi in metri."""
    lat1, lon1 = graph.nodes[node1]['y'], graph.nodes[node1]['x']
    lat2, lon2 = graph.nodes[node2]['y'], graph.nodes[node2]['x']
    return geodesic((lat1, lon1), (lat2, lon2)).meters

def incidenti_medi_per_km(graph):
    """Calcola il numero medio di incidenti per chilometro nel grafo."""
    total_incidents = sum(int(data.get("incidenti", 0)) for u, v, data in graph.
 ↪edges(data=True))
    total_length = sum(float(data.get("length", 1)) for u, v, data in graph.
 ↪edges(data=True)) / 1000  # Converti in km
    return total_incidents / total_length if total_length > 0 else 0

def lunghezza_media_archi(graph):
    """Calcola la lunghezza media degli archi nel grafo."""
    edge_lengths = [float(data.get("length", 1)) for u, v, data in graph.
 ↪edges(data=True)]
    return sum(edge_lengths) / len(edge_lengths) if edge_lengths else 1

def min_incident_rate_per_km(graph):
    min_rate = float('inf')
    for u, v, data in graph.edges(data=True):
        length_km = float(data.get("length", 1)) / 1000  # Converti in km
        incidents = int(data.get("incidenti", 0))
        if length_km == 0:
```

```
            continue  # Evita divisione per zero
        rate = incidents / length_km
        if rate < min_rate:
            min_rate = rate
    return min_rate if min_rate != float('inf') else 0

def heuristic_1(graph, node, goal):
    """Calcola l'euristica h(n) per il nodo corrente basata su distanza e␣
 ↪incidenti."""
    distanza = geodetic_distance(graph, node, goal)
    incidenti_km = incidenti_medi_per_km(graph)
    lunghezza_media = lunghezza_media_archi(graph)
    return distanza * (incidenti_km / lunghezza_media)

def heuristic_2(graph, node, goal, min_rate):
    distance_meters = geodetic_distance(graph, node, goal)
    distance_km = distance_meters / 1000
    return distance_km * min_rate
```

3.1 A*

L'algoritmo **A*** è un algoritmo di ricerca informata che esplora un grafo per trovare il percorso ottimale tra un nodo di partenza e un nodo di arrivo, minimizzando un costo predefinito. Nel contesto di questo progetto, l'obiettivo dell'algoritmo A* è ***minimizzare il numero di incidenti lungo il percorso***.

L'algoritmo funziona come segue:

- **Coda di priorità:** L'algoritmo utilizza una coda di priorità per esplorare i nodi, ordinata in base al valore di $f(n) = g(n) + h(n)$:
  - ***g(n)*** è il costo accumulato del percorso fino al nodo corrente, che in questo caso corrisponde al numero di incidenti lungo il percorso.
  - ***h(n)*** è la *funzione euristica* che stima il costo rimanente per raggiungere il nodo di arrivo. In questo caso, l'euristica è calcolata come una stima basata sul minimo tasso di incidenti per km nel grafo, utilizzando una combinazione di distanza e incidenti attesi.
- **Esplorazione dei nodi:** Ad ogni iterazione, il nodo con il valore $f(n)$ più basso viene estratto dalla coda di priorità. Se il nodo corrente è il nodo di arrivo, il percorso viene restituito come soluzione. Se il nodo è già stato visitato con un costo inferiore o uguale, viene ignorato per evitare percorsi peggiori.
- **Espansione dei nodi:** Se il nodo corrente non è il nodo di arrivo, l'algoritmo espande i suoi vicini (nodi adiacenti) e calcola il nuovo costo g(n) per ciascun vicino. Se il nuovo costo è migliore del costo precedentemente registrato per quel nodo, il vicino viene aggiunto alla coda di priorità.
- **Memorizzazione dei percorsi esplorati:** Durante l'esecuzione, l'algoritmo salva tutti i

percorsi esplorati in una lista explored_paths, che consente di monitorare la ricerca.
Finitura: Quando il nodo di arrivo viene trovato, l'algoritmo termina restituendo il percorso con il minor numero di incidenti. Inoltre, vengono restituiti anche il tempo di esecuzione, la memoria utilizzata, e la profondità massima raggiunta durante la ricerca.

[11]:
```python
def a_star_search(graph, start, goal, min_rate):
    """Implementazione dell'algoritmo A* per minimizzare il numero di incidenti.
    """
    priority_queue = [(0, start, 0, [start])]  # (f(n), nodo, g(n), percorso)
    visited = {}
    explored_paths = []  # Ora è una lista di percorsi esplorati
    start_time = time.time()
    memory_before = sys.getsizeof(priority_queue) + sys.getsizeof(visited)
    max_depth = 0

    while priority_queue:
        f, node, g, path = heapq.heappop(priority_queue)  # Estrai il nodo con␣
    ↪il miglior f(n)
        explored_paths.append(path)  # Salva il percorso esplorato
        max_depth = max(max_depth, len(path) - 1)

        if node == goal:
            execution_time = time.time() - start_time
            memory_after = sys.getsizeof(priority_queue) + sys.
    ↪getsizeof(visited)
            memory_used = memory_after - memory_before
            return path, g, explored_paths, execution_time, memory_used,␣
    ↪max_depth  # Restituisce i percorsi esplorati

        if node in visited and g >= visited[node]:
            continue  # Ignora percorsi peggiori

        visited[node] = g  # Memorizza il miglior costo trovato per questo nodo

        for neighbor in graph.neighbors(node):
            edge_data = graph.get_edge_data(node, neighbor)

            if isinstance(edge_data, dict):
                edge_data = min(edge_data.values(), key=lambda d: int(d.
    ↪get("incidenti", float('inf'))))

            edge_cost = int(edge_data.get("incidenti", 0))
            new_g = g + edge_cost
            # h = heuristic_1(graph, neighbor, goal) # Euristica 1
            h = heuristic_2(graph, neighbor, goal, min_rate)
            f = new_g + h
```

```
            heapq.heappush(priority_queue, (f, neighbor, new_g, path +␣
 ↪[neighbor]))

    execution_time = time.time() - start_time
    memory_after = sys.getsizeof(priority_queue) + sys.getsizeof(visited)
    memory_used = memory_after - memory_before
    return None, float('inf'), explored_paths, execution_time, memory_used,␣
 ↪max_depth  # Ora explored_paths è una lista


# Caricare il grafo con il numero di incidenti sugli archi
G = ox.load_graphml("Dati\\rete_bari_incidenti.graphml")

partenza = 270659688
arrivo = 1481415203

# Heuristic 2
min_rate = min_incident_rate_per_km(G)
shortest_path, min_incidents, explored_paths, exec_time, memory_used, max_depth␣
 ↪= a_star_search(G, partenza, arrivo, min_rate)

if shortest_path:
    print(f"Percorso con il minor numero di incidenti trovato: {shortest_path}")
    print(f"\nNumero totale di incidenti lungo il percorso: {min_incidents}")
    print(f"\nNumero di percorsi visitati: {len(explored_paths)}")
    print(f"Tempo di esecuzione: {exec_time:.4f} secondi")
    print(f"Memoria utilizzata: {memory_used} byte")
    print(f"Profondità massima raggiunta: {max_depth}")
else:
    print("Nessun percorso trovato.")
    print(f"Numero di percorsi visitati: {explored_paths}")
    print(f"Tempo di esecuzione: {exec_time:.4f} secondi")
    print(f"Memoria utilizzata: {memory_used} byte")
    print(f"Profondità massima raggiunta: {max_depth}")
```

```
Percorso con il minor numero di incidenti trovato: [270659688, 270388628,
322548994, 322548996, 12083787147, 569212252, 3860917204, 3841272949,
3860940020, 330076413, 279650482, 1483634369, 1483634284, 279655605, 280884748,
279380962, 339607009, 279655525, 10680697917, 316572227, 10680697906, 279655521,
9602691583, 1014703348, 1249973250, 4395827572, 5395447618, 5395447592,
5395448236, 5395447590, 330932109, 1459966804, 298502133, 329994613, 330003439,
1481415203]

Numero totale di incidenti lungo il percorso: 123

Numero di percorsi visitati: 6239
Tempo di esecuzione: 0.8437 secondi
```

```
Memoria utilizzata: 152208 byte
Profondità massima raggiunta: 129
```

3.1.1 Valutazione e Confronto dei Risultati in Base all'Euristica Utilizzata

Di seguito viene effettuato un confronto tra i risultati ottenuti utilizzando due euristiche differenti nell'algoritmo **A***: l'**euristica h1** (non ammissibile) e l'**euristica h2** (ammissibile). I risultati sono analizzati in base a **numero di incidenti**, **numero di percorsi visitati**, **tempo di esecuzione**, **memoria utilizzata** e **profondità massima raggiunta**.

Metriche	**Euristica h1 (Non Ammissibile)**	**Euristica h2 (Ammissibile)**
Percorso con il minor numero di incidenti trovato	[270659688, 270388628, 322548994, 322548996, 322548997, 270388924, 270388545, 330040330, 298504324, 298504314, 2882564567, 298504321, 10604170442, 11476336896, 270434053, 329994353, 270433936, 1108920034, 320970982, 320970994, 270654807, 270655162, 329988598, 329988600, 270655174, 270654641, 1481415203]	[270659688, 270388628, 322548994, 322548996, 12083787147, 569212252, 3860917204, 3841272949, 3860940020, 330076413, 279650482, 1483634369, 1483634284, 279655605, 280884748, 279380962, 339607009, 279655525, 10680697917, 316572227, 10680697906, 279655521, 9602691583, 1014703348, 1249973250, 4395827572, 5395447618, 5395447592, 5395448236, 5395447590, 330932109, 1459966804, 298502133, 329994613, 330003439, 1481415203]
Numero totale di incidenti lungo il percorso	211	123
Numero di percorsi visitati	31	6239
Tempo di esecuzione	1.6824 secondi	0.8437 secondi
Memoria utilizzata	1352 byte	152208 byte
Profondità massima raggiunta	26	129

Osservazioni e Analisi

1. **Percorso con il minor numero di incidenti**:
   - Con **l'euristica h1** (non ammissibile), l'algoritmo ha trovato un percorso con **211 incidenti**.
   - Con **l'euristica h2** (ammissibile), l'algoritmo ha trovato un percorso con **123 incidenti**, risultando quindi un percorso più sicuro in termini di incidenti.

2. **Numero di percorsi visitati**:
   - Con **h1**, l'algoritmo ha esplorato **31 percorsi**.
   - Con **h2**, il numero di percorsi esplorati è stato significativamente maggiore: **6239 percorsi**. Nonostante ciò, **l'euristica h2** ha portato a una soluzione con meno incidenti, suggerendo che la qualità dell'euristica ammortizza il maggiore numero di percorsi esplorati.
3. **Tempo di esecuzione**:
   - L'algoritmo con **h2** ha impiegato **0.8437 secondi**, che è più veloce rispetto a **1.6824 secondi** per **h1**.
   - Nonostante un numero maggiore di percorsi esplorati con **h2**, il tempo di esecuzione è inferiore, suggerendo che l'euristica **h2** è più efficiente nel guidare la ricerca verso soluzioni ottimali.
4. **Memoria utilizzata**:
   - L'algoritmo con **h1** ha consumato **1352 byte** di memoria, mentre con **h2** la memoria utilizzata è stata **152208 byte**, significativamente maggiore. Questo potrebbe essere dovuto al numero maggiore di percorsi esplorati e alla necessità di memorizzare informazioni aggiuntive durante l'esecuzione con l'euristica **h2**.
5. **Profondità massima raggiunta**:
   - L'algoritmo con **h1** ha raggiunto una profondità massima di **26**.
   - Con **h2**, la profondità massima è stata molto maggiore, **129**, indicando che l'algoritmo ha esplorato una porzione molto più ampia del grafo per trovare la soluzione ottimale.

Conclusione
L'analisi dei risultati ha mostrato che l'**euristica h2**, pur richiedendo più risorse in termini di memoria e tempo di esecuzione, ha prodotto il percorso più sicuro, riducendo il numero di incidenti. Al contrario, **l'euristica h1**, sebbene più veloce e meno costosa in termini di risorse, ha portato a soluzioni subottimali con più incidenti.