POLITECNICO DI MILANO

SCUOLA DI INGEGNERIA INDUSTRIALE E DELL'INFORMAZIONE

A.A. 2024-2025

PROGETTO DI INGEGNERIA INFORMATICA

HPC processing for supporting Car Navigation systems

A cura di: Leonardo Sinibaldi e Giorgia Savo

Prof. referente: Gianluca Palermo

Milano, May 28, 2025

Indice

Elenco delle figure

1. Introduzione

1.1. Obiettivo

L'obiettivo del progetto sviluppato è l'elaborazione di un sistema ad alte prestazioni (HPC - High Performance Computing) in grado di **supportare** in modo efficiente i moderni **sistemi di navigazione per veicoli**.
Il progetto mira a gestire e processare i dati provenienti da sorgenti esterne, nel nostro caso OpenStreetMap, per offrire servizi di routing intelligenti e ottimizzati su diverse scale geografiche.
In particolare, dati un sorgente e una destinazione, lo scopo del progetto è quello di fornire, oltre che lo *shortest-path*, ulteriori soluzioni di percorso sufficientemente diverse tra loro per essere considerate effettivamente delle strade alternative.

1.2. Teconologie utilizzate

Per sviluppare il progetto sono stati utilizzati diversi linguaggi e librerie , in particolare:

- **ARLIB** : libreria sviluppata da Leonardo Arcari, per l'utilizzo di algoritmi di esplorazione dei grafi per trovare diversi percorsi;
- **C++** : linguaggio utilizzato per la stesura del backend che prevede l'utilizzo della libreria precedentemente citata;
- **Python** : utilizzato invece per il frontend, sviluppato con l'apertura di una semplice applicazione che chiede all'utente di inserire sorgente, destinazione e alcuni parametri per il calcolo delle *alternative routes*, permette la visualizzazione di quest'utlime dopo averle ricevute dal backend.

1.3. Fasi del progetto

Il progetto sviluppato ha previsto 3 fasi principali che ci hanno portato ad approfondire e realizzare al meglio l'applicazione, in particolare:

- **Analisi di *papers***: Una prima fase di analisi di diversi articoli forniti dal professor Palermo e di alcuni trovati in rete da noi.
  Durante questa fase ci siamo anche documentati sulla libreria ARLIB precedentemente citata.
- **Stesura del codice** : si è poi passati alla stesura del codice per quanto riguarda entrambe le parti di backend (in C++) e frontend (in Python)
- **Analisi delle computazioni**: infine, una volta completato il codice, l'obiettivo si è spostato sull'analisi dei tempi e delle prestazioni dei vari algoritmi mediante l'utilizzo di un sistema HPC.

Le fasi precedentemente citate saranno ulteriormente approfondite nei prossimi capitoli.

2. Analisi degli articoli

In questa prima fase, durata circa 3 settimane, ci siamo concentrati sulla comprensione, studio e rielaborazione di diversi articoli riguardanti algoritmi di computazione per la ricerca di percorsi alternativi sufficientemente dissimili su grafi.
Uno degli obiettivi del progetto è stato trovare, analizzare e valutare degli algoritmi esatti ed euristici per il calcolo di percorsi alternativi con il fine di ridurre al minimo il costo computazionale mantenendo un buon grado di qualità dei risultati. In particolare infatti, citando proprio uno degli articoli in esame, *we consider the problem of tracing alternative paths from a source node s to a target node t in G, with edge weight or cost function:* $E \rightarrow R^{+}$. *The essential goal is to obtain sucfficiently different paths with* ***optimal or near optimal cost***. [1]

La nostra attenzione si è perciò rivolta a tre grandi famiglie di algoritmi:

- **Algoritmi per il PERCORSO più BREVE**
- **Algoritmi ESATTI**
- **Algoritmi EURISTICI**

2.1. Shortest path

Algoritmi ideati per trovare il cammino minimo, ne esistono di diverse tipologie. Per trovare il primo cammino, ogni algoritmo di ricerca di percorsi alternativi utilizza una di queste metodologie, in modo tale che la prima soluzione restituita sia quella ottima, le successive invece sono considerate *althernative paths*.
Tutti gli algoritmi citati in questa sezione si trovano nel paper [1]

2.1.1. Forward Dijkstra

Costruisce un **albero completo dei cammini piu brevi** dalla radice nodo s tramite l'algoritmo di BFS, esplorando i nodi in G in ordine crescente di distanza da s. L'ordine di esplorazione puo essere tenuto sotto controllo da una coda di priorita Q: in ogni iterazione il nodo u a distanza minima d(s, u) è rimosso da Q e i suoi archi uscenti sono *rilassati*.
For any outgoing edge of u, $e = (u, v) \in E$*, if* $d(s, u) + w(e) < d(s, v)$ *then it sets* $d(s, v) = d(s, u) + w(e)$ *and* $pred(v) = u$*.* [1]
Quindi per ogni arco=(u,v), se passando per u si trova un percorso più breve verso v, **si aggiorna la distanza tentativa di v con il nuovo valore più basso e si memorizza u come predecessore di v**. Una volta che il nodo vene rimosso dalla coda, significa che la distanza minima è settata e quindi non può più essere cambiata e quel nodo stabilizzato non verrà piu riesaminato. L'algoritmo termina quando la coda sarà vuota.
Forward: perche l'algoritmo si espande dalla sorgente s in avanti verso gli altri nodi.

2.1.2. Backward Dijkstra

Si esplorano i nodi a partire dall'ordine inverso, quindi da t, attraversando gli archi entranti dei nodi nella direzione opposta. In questa versione sono registrati i successori dei nodi.
È spesso usata questa versione per ridurre l'esplorazione di nodi che non appartengono a un percorso piu breve dei cammini.

2.1.3. Bidirectional Dijkstra

Applica contemporaneamente:

- Forward Dijkstra → da s a t
- Backward Dijkstra → da t a s

Sono applicati alternativamente finche non si incontrano. Combina i percorsi parziali s→v e v→t dai punti di incontro, mantenendo il percorso con costo minimo $w(s, v) + w(v, t)$.

Termina quando la somma delle distanze minime nei due fronti supera la distanza d(s,t).

2.1.4. A*

Modifica il Dijkstra classico con l'uso di una funzione euristica $h_t(u)$ che stima il costo minimo dal nodo corrente u al nodo obiettivo t modificando la coda di priorita Q con questa funzione.
Questa condizione garantisce che **A* esplori solo percorsi validi e produca la soluzione ottimale** *"indirizzando"* la ricerca verso la destinazione.
Più il costo minimo è piccolo più la ricerca diventa rapida rischiando però ridurre l'accuratezza del risultato.
Proprietà richieste dalla funzione: monotonia $h_t(u) \leq w(u,v) + h_t(v), \forall (u,v) \in E$.

2.2. Algoritmi ESATTI

Si basano sull'esplorazione sistematica di tutte le possibili soluzioni (o parte sufficiente di esse) per garantire di individuare la **soluzione ottimale**. Dalla lettura abbiamo individuato inoltre pro e contro:

- **Svantaggi**: non tengono tanto conto di risorse computazionali quali tempo e memoria
- **Vantaggi**: garantiscono l'individuazione di risorse ottimali

2.2.1. OnePass

Traverses the road network expanding path from the source node s ***while pruning partially expanded paths that cannot lead to a result as early as possible.*** [2]

L'obiettivo di OnePass quindi è quello di esplorare il grafo, come farebbe un qualsiasi algoritmo classico, tuttavia si pone l'obiettivo di non esplorare *"inutilmente"* molti percorsi subottimali. Quindi OnePass elimina (`prune`) *"in anticipo"* quei percorsi che sicuramente non porteranno a una soluzione ottimale. Se un percorso parziale **non**

potrà mai arrivare a una destinazione **più velocemente** di un altro già trovato o potenziale, viene **scartato** subito. In pratica, si evita di "perdere tempo" su strade che, anche se esplorate completamente, non porteranno mai a un risultato migliore.
L'algoritmo si basa su due lemmi fondamentali:

- **Lemma 1**: Dati due percorsi p e p , e S=p p =e_1, ..., e l'insieme degli archi condivisi, la similarità unidirezionale tra p e p è definita come:

  $$\overrightarrow{\text{Sim}}(p, p') = \sum_{e_i \in p \cap p'} \overrightarrow{\text{Sim}}(\langle e_i \rangle, p')$$

  dove e è il sottopercorso di p contenente solo l'arco e .

- **Lemma 2**: Sia PLO l'insieme corrente dei percorsi risultati. Se p è un nuovo percorso alternativo, per essere considerato valido deve valere:

  $$\text{Sim}(p_{\text{sub}}, p_i) \leq \theta$$

  per ogni sottopercorso p_sub di p e per ogni percorso p in PLO.

In altre parole: *un nuovo percorso viene accettato solo se non è troppo simile a nessuno dei percorsi già selezionati, anche considerando ogni suo sottopercorso*.

Notiamo quindi immediatamente che **se la one-way similarity di un cammino viola la soglia** allora anche **tutte le estensioni** del cammino lo faranno e quindi le soluzioni verranno considerate ***infeasible***.
I cammini nell'algoritmo sono esaminati in ordine di lunghezza crescente.
Complexity analysis: nel peggiore dei casi enumeriamo tutti i possibili (s → t) cammini e quindi la complessità sara O(poly(K)) dove E(K)=((| N | -2)!d) con E(K) indichiamo il valore atteso del numero di cammini K in un grafo.

2.2.2. Multipass

Traverses the road network expanding path from the source node s while pruning partially expanded paths that cannot lead to a result as early as possible, ***it employs a second prunin criterion that aims at reducing the search space by avoiding the expansion of non-promising paths***. [2]

Quindi viene implementato esattamente allo stesso modo di OnePass, con la differenza di applicare un ulteriore criterio di *"potatura"*, che ha lo scopo di ridurre lo spazio di ricerca, **evitando l'espansione di percorsi poco promettenti**.

A differenza di OnePass, MultiPass può **attraversare la rete più volte**, ma la sua complessità non dipende direttamente dal numero di percorsi esistenti (s → t) che in un grafo stradale risulta essere molto elevato.

L'algoritmo si basa su tre lemmi fondamentali:

- **Lemma 1**: precedentemente discusso 2.2.1
- **Lemma 2**: anche questo nella sezione precedente 2.2.1
- **Lemma 3**: Sia P un insieme di percorsi da un nodo sorgente s a un nodo target t, e siano p , p due percorsi da s a un nodo intermedio n. Se:

  $$\ell(p_j) > \ell(p_i) \quad \text{e} \quad \forall p \in P : \text{Sim}(p_i, p) \leq \text{Sim}(p_j, p)$$

  allora 'p ' non può far parte del più corto percorso alternativo a 'P', e si scrive:

  $p_i \prec_P p_j$

L'algoritmo visita i percorsi in ordine di lunghezza crescente, espandendo ogni percorso p(s → n) solo se rispetta le condizioni imposte dai tre lemmi. Questo approccio consente di filtrare i percorsi non promettenti in modo efficace, garantendo risultati ottimali.
Complexity analysis: L'algoritmo presenta una complessita di $\mathcal{O}(k \cdot |N|^2 \cdot (\theta L)^{2k})$ dove: k è il numero di percorsi richiesti, |N| è il numero di nodi della rete, L la somma dei pesi associati agli archi appartenenti al percorso trovato (nel caso di una rete stradale si traduce bene con la lunghezza del percorso) e θ la similarità.

2.3. Algoritmi EURISTICI

Non garantiscono di trovare la soluzione ottimale, ma mirano a trovare una soluzione sufficientemente buona in tempi più rapidi, specialmente quando il problema è complesso e la ricerca della soluzione ottimale è impraticabile.

- **Svantaggi**: non è sempre la soluzione ottima
- **Vantaggi**: computazioni molto efficienti

2.3.1. OnePass+

In contrast to MULTIPASS each each time a new path is added to the result set PLO, OnePass+ does not restart the traversal like MultiPass, but continues in a similar fashion to OnePass, thus ***traversing the network only once*** [2]

OnePass+ perciò, come Multipass, utilizza tutti e tre i lemma, tuttavia ha la caratteristica di attaraversare la rete una sola volta, di conseguenza questo lo porta ad essere un algoritmo con un set di soluzioni trovate molto vicine alla soluzione ottima.
Complexity analysis: $\mathcal{O}(|N|^2 \cdot (\theta L)^{2k})$, anche nella complessità e facile notare la differenza con Multipass, non è infatti presente il termine *k* eliminato in quanto OnePass+ attraversa la rete solo una volta.

2.3.2. SVP+

The main idea behind SVP+ is similar to the baseline method for computing kSPwLO queries discussed. However, ***instead of iterating over all possible (s→t) paths, SVP+ iterates over the much smaller set of single-via paths.*** [2]

Si implementa il concetto del *single-via paths*.
Definizione di single-via path: Data una rete stradale G=(N, E), un nodo di partenza s e un nodo di arrivo t, il ***single-via path*** di ciascun nodo $n \in N$ è la concatenzazione dello *shortest-path* $p_{sp}(s \rightarrow n)$ e $p_{sp}(n \rightarrow t)$. L'obiettivo di SVP+ è trovare un set di k single-via path così che:

- sia sempre consigliato il percorso piu breve possibile
- ogni single-via path trovato è sufficientemente dissimile dagli altri
- tutti i k single-via paths sono il più breve possibile.

L'algoritmo termina quando abbiamo almeno k cammini oppure non esistono più single-via paths da esaminare.
Complexity analysis: questo algoritmo nel caso pessimo usa l'algoritmo di Dijkstra due volte per ogni vertice del grafo, è inoltre necessario un tempo lineare rispetto a k per decidere se accettare o meno un percorso calcolato. La complessità risultate è dunque: $O(|N|k + |E| + |N|log|N|)$.

2.3.3. ESX

*ESX computes kSPwLO by executing shortest path searches while **progressively excluding edges from the road network**.* [2]

É un algoritmo diverso da tutti quelli analizzati precedentemente, si basa infatti sul *tagliare* archi dalla rete stradale.
Sono due gli aspetti importanti da sottolineare:

- **l'ORDINE** in cui i cammini sono rimossi dalla rete della strada Soluzione: se dopo non riusciamo a trovare un nuovo cammino allora ESX ri-inserisce il nodo nella rete e lo marca come non-rimuovibile
- **il MANTENIMENTO** della connessione della rete

Complexity analysis: $O(k|N|(|N|log|N| + |E|))$

Strategie di RIMOZIONE: come capire quale cammino togliere

- **smallest/largest edge weight**: usa il peso per selezionare. Si dà priorità agli archi con peso minore (MinW variant, esiste anche la MaxW variant) o peso maggiore:
  - MaxW variant → il prossimo cammino sarà meno simile a quello gia computato e l'algoritmo terminerà piu velocemente, TUTTAVIA ci sono alte possibilità di mancare path alternativi

  - MinW variant → ci aspettiamo cammini con peso maggiore, si incrementano le iterazioni in questo modo e quindi l'overall runtime

- **minimum/maximum stretch**: partendo innanzitutto dalla definizione di stretch, possiamo dire che: dato un arco $e = (n_i, n_j)$, considerato p il cammino più corto da n_i a n_j computato escludendo l'arco e. Lo **stretch** di un arco è la differenza tra la lunghezza del percorso p e la lunghezza dell'arco e escluso, quindi $stretch(e) = |(p)-(e)|$.
  Anche in questo caso abbiamo delle varianti:

  - MaxS variant → puo causare deviazione piu facilmente, portando a cammini meno simili a uqello orecedente e trova soluzione velocemente
  - MinS variant → si esaminano piu cammini, questo incrementa overall runtime, MA si incrementa probabilita di contenere nel set del result lo shortest path

last/most local shortest paths: basato sulla edge betweenness MaxP variant → l'intuizione sta nel fatto che più un arco è attraversato dai cammini e più la sua rimozione causerà detour MinP variant → se implementiamo questo cresce la possibilità di computare path alternativi che sono piu corti in media ma il costo overall aumenta.

2.4. Algoritmi di P-DISPERSIONE

La famiglia di algoritmi ora analizzata si basa su quello che viene chiamato **"modello" di p-dispersione**, che, citando l'articolo analizzato, significa:

The p-dispersion model is a facility location model which considers the porblem of selecting a dispersed subset of a given set of points in some space so that the minimum distance between pairs of selected points is maximized [3]

Come fare: si applica un algoritmo di p-dispersione a un insieme di cammini alternativi generati da un k-shortest path algorithm (già discusso sopra 2.3).

2.4.1. Penalty

L'algoritmo Penalty o in modo più esteso l'Iterative Penalty Method si basa sull'**applicazione iterativa** di algoritmi di ricerca di cammini minimi ⇒ applicazione di una penalità su tutti i collegamenti del percorso più breve ⇒ scoraggiamento selezione sempre dello stesso cammino.

Vi sono diversi elementi da analizzare per questa categoria di algoritmi:

- **Penalized units**: le unità penalizzate sono:
  - collegamenti
  - nodi
  - collegamenti e nodi
- **Penalty structure**: come avviene la penalità
  - additive penalty
  - multiplicative penalty
- **Penalty magnitude**: è il peso della penalizzazione che applichiamo
- **Penalized paths**: sceglie quali percorsi penalizzare

Vantaggio: SEMPLICITA
Svantaggio: non valuta la QUALITÀ del set di cammini che vengono generati in termini di differenza spaziale e lunghezza dei cammini.
Vi sono 3 importanti misure per valutare la qualità di un set di cammini:

- la lunghezza media
- la dissimilarità media tra coppie di cammini
- la minima dissimilarità in assoluto

Similarity: $S(P_i, P_j) = [L(P_iP_j)/L(P_i) + L(P_iP_j)/L(P_j)]/2$
Dissimilarity: $D(P_i, P_j) = 1 - S(P_i, P_j)$

Per quanto riguarda l'IPM analizzato nel paper e utilizzato anche nel nostro progetto, le caratteristiche sono:

- penalised units: links
- penalty structure: multiplicative penalty
- penalty computation:
  - basato sulla impedenza corrente (CI)
  - basato sulla impedenza originale (OI)
- penalty magnitude: sono stati usati sia pesi piccoli sia grandi
- penalized paths: solo agli archi del cammino più recente

2.5. Plateau

Infine tra gli algoritmi di ricerca di cammini alternativi si distingue anche il **metodo dei Plateau** [1]. Fornisce dei cammini alternativi **connettendo coppie di cammini piu corti**

- da s a v
- da v a t

Il nodo v è selezionato sulla base del se è appartenente a un **plateau**, per avere alternative a bassa sovrapposizione di cammini.

Cos'è un plateau?

Sono cammini semplici $\overline{P} \subseteq Pst$ consistenti di più di un nodo successivo, con la proprietà che $d_s(u) + d_t(u) = d_s(v) + d_t(v) \forall u, v \in P$
Quindi rappresentano delle **zone piatte nelle distanze tra s e t**. Cioè qualunque distanza totale dalla sorgente s al target t tramite qualsiasi nodo nel plateau è costante. Un plateau è quindi una **regione del grafo dove diversi nodi possono potenzialmente contribuire a cammini alternativi senza cambiare il costo totale**.

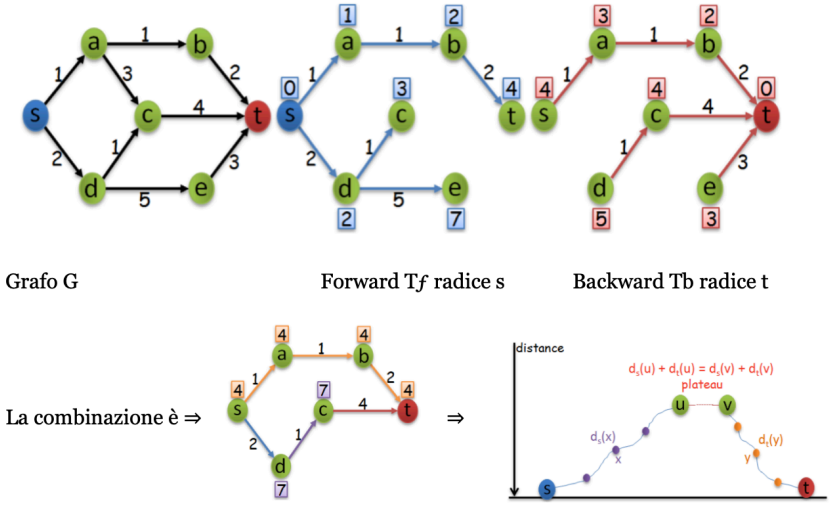

Figura 2.1. rappresentazione grafica del metodo dei plateau

Dato che i plateau di solito sono troppi, c'è uno stadio di filtraggio che è usato per selezionare il migliore di questi.

[1]

3. Implementation

Dopo l'analisi approfondita dei papers, l'attenzione si è quindi spostata sul codice. L'obiettivo è stato infatti quello di creare un'applicazione, completa di frontend e backend, che potesse effettivamente mettere in pratica gli algoritmi precedentemente studiati in modo tale da verificare e calcolare percorsi alternativi da un nodo source s a un nodo target t.
L'intero progetto è sostanzialmente divisibile in due macrosezioni: frontend e backend.

3.1. C++ backend

Il backend del progetto è implementato come un server C++ progettato per gestire richieste di calcolo intensivo. Opera esponendo un'interfaccia di comunicazione tramite socket TCP su una porta configurabile (10714 di default). Le interazioni tra client e server avvengono mediante lo scambio di messaggi strutturati, composti da una parte di intestazione (`header`) e una parte contenente i dati effettivi (`body`). L'header fornisce informazioni preliminari necessarie per la corretta interpretazione del body, come la sua dimensione. I dettagli specifici del protocollo di comunicazione sono approfonditi nella sottosezione 3.1.2.

Una richiesta di computazione inviata dal client al server contiene i parametri fondamentali per l'elaborazione: i nodi di origine e destinazione del percorso (identificati tramite OSMID), il numero desiderato di percorsi alternativi (k) e un fattore di dissimilarità (θ). La struttura di questo payload è illustrata in Figura 3.1. Per la prima richiesta relativa a una nuova area geografica, il client deve includere anche i dati del grafo stradale in formato `.graphml`. Per richieste successive sullo stesso grafo, è sufficiente inviare solo i parametri di calcolo (costituiti da 28 byte), ottimizzando così la quantità di dati trasmessi.

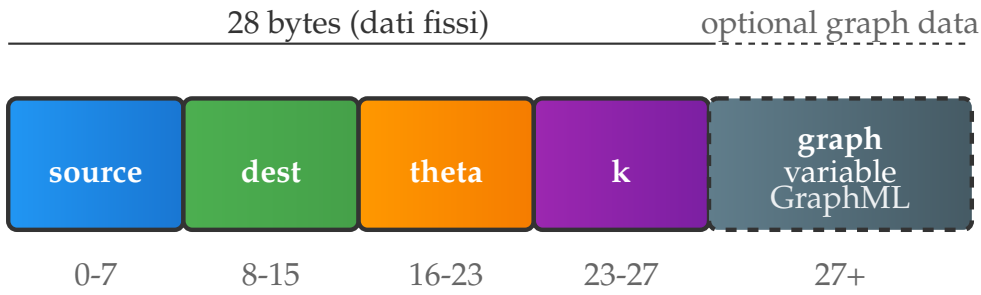

Figura 3.1. Struttura del messaggio di richiesta contenente i parametri e, opzionalmente, il grafo.

In risposta, il server esegue gli algoritmi di calcolo richiesti, come dettagliato nella sottosezione 3.1.1. Un aspetto chiave del backend è la capacità di inviare i risultati al client in modo incrementale: non appena un algoritmo individua un percorso valido, questo viene immediatamente trasmesso. Ogni messaggio di risposta, la cui struttura è mostrata in Figura 3.2, contiene: il nome dell'algoritmo che ha generato il percorso (`alg`), un identificativo numerico del percorso (`count`), la sua lunghezza effettiva (`length`) e la sequenza di identificativi OSMID (`osmid_path`) dei nodi che lo compongono. Al termine di tutte le computazioni per una data richiesta, il server invia un messaggio di notifica finale.

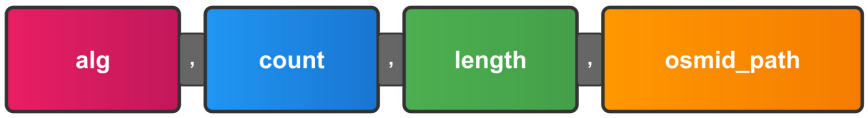

Figura 3.2. Struttura del messaggio di risposta inviato dal server per ogni percorso trovato.

3.1.1. Algoritmi

Il nucleo elaborativo del backend è incapsulato nella classe `Engine`, che gestisce l'intero flusso di lavoro: dalla ricezione e interpretazione delle richieste del client, alla costruzione del grafo (se fornito o se è il primo utilizzo), fino all'esecuzione parallela degli algoritmi per la ricerca di percorsi alternativi. Una volta che una richiesta è stata

validata e il grafo è disponibile, il metodo `Engine::runAlg` si occupa di avviare il processo di calcolo.

Preliminarmente, all'interno di `runAlg`, gli OSMID dei nodi sorgente e destinazione vengono convertiti nei corrispondenti descrittori di vertice interni al grafo, utilizzando la funzione di utilità `utils::find_vertex_by_osmid`. Assicurata la validità di tali vertici, il sistema avvia l'esecuzione concorrente di tre distinti algoritmi di ricerca. Ogni algoritmo viene eseguito in un thread separato (`std::thread`), consentendo una parallelizzazione efficace dell'elaborazione e una potenziale riduzione del tempo complessivo di attesa per l'utente, dato che i risultati possono arrivare da algoritmi con diverse velocità di convergenza.

Gli algoritmi impiegati sono forniti dalla libreria `arlib` sviluppata da Leonardo Arcari. e comprendono:

- `onepass_plus` 2.3.1
- `esx` 2.3.3
- `penalty` 2.4.1

Ciascun thread esegue la funzione `Engine::get_alternative_routes`, specificando quale dei tre algoritmi utilizzare. Questa funzione agisce come wrapper per `utils::run_alt_routing`, che a sua volta si interfaccia con le routine specifiche della libreria `arlib`, fornendo loro il grafo, la mappa dei pesi, i parametri s, t, k, θ. Un meccanismo di callback è stato integrato con `arlib`: durante l'elaborazione, non appena un algoritmo identifica un percorso valido, viene invocato il metodo `Engine::savePath`. Questo permette l'invio incrementale dei risultati al client.

Il metodo `Engine::savePath` è quindi cruciale per la comunicazione dei risultati. Riceve il percorso trovato (come `std::vector<Edge>`), un contatore progressivo specifico per l'algoritmo (`count`), e il nome dell'algoritmo (`alg`). Successivamente, formatta queste informazioni in una stringa che include anche la lunghezza del percorso e la lista degli OSMID dei nodi (ottenuta tramite `utils::get_osmid_path`), come descritto nella struttura di risposta (Figura 3.2). Questa stringa viene incapsulata in un oggetto `message` e trasmessa al client tramite `NetworkProvider::send`. Per gestire l'accesso concorrente alla risorsa di rete da parte dei vari thread, l'operazione di invio è protetta da un `std::mutex` (`m_resultsMutex`). Una volta che tutti i thread dedicati agli algoritmi hanno completato la loro esecuzione (sincronizzati tramite `join()`), il server

invia un messaggio finale `"COMPUTATION_DONE"` al client, segnalando la conclusione di tutte le elaborazioni per la richiesta corrente.

3.1.2. Network

La gestione della comunicazione di rete è interamente demandata alla classe `NetworkProvider`, la quale realizza un server TCP utilizzando le funzionalità offerte dalla libreria Boost.Asio. All'avvio, come specificato nel suo costruttore `NetworkProvider::NetworkProvider(const int port)`, il server si pone in ascolto sulla porta TCP designata (10714 come valore predefinito), pronto ad accettare connessioni entranti dai client.

Il ciclo di interazione con un client ha inizio quando il metodo `NetworkProvider::connect()` stabilisce con successo una connessione, bloccando l'esecuzione fino a quel momento. Una volta connesso, il server è pronto per il protocollo di scambio messaggi.

La ricezione di un messaggio dal client, gestita dal metodo `NetworkProvider::receive()`, si articola in due fasi principali, implementate rispettivamente da `receiveHeader` e `receiveBody`:

1. **Ricezione dell'Header**: Il server si mette in attesa di 4 byte (`m_headerBuffer`). Questi byte costituiscono un intero a 32 bit, trasmesso in formato big-endian, che specifica la dimensione totale del messaggio in arrivo (header + body). Dopo aver ricevuto e decodificato questa dimensione, il server alloca lo spazio necessario per il corpo del messaggio (`m_bodyBuffer`) e invia una stringa di conferma `"ok"` al client.
2. **Ricezione del Body**: Successivamente all'invio dell'`"ok"`, il server attende di ricevere il corpo del messaggio, la cui dimensione è stata determinata nella fase precedente. I dati vengono letti direttamente nel `m_bodyBuffer`. Al completamento della ricezione dell'intero corpo, il server invia un'ulteriore conferma `"ok"` al client.

I dati così ricevuti (header e body) sono poi utilizzati per costruire un oggetto `message` tramite il metodo `buildMessage`.

Analogamente, l'invio di messaggi dal server al client, effettuato tramite `NetworkProvider::send(const message message)`, adotta un protocollo a due fasi con conferma, gestito dai metodi ausiliari `sendHeader` e `sendBody`:

1. **Invio dell'Header**: Il server trasmette per primi 4 byte che rappresentano la dimensione totale del messaggio in uscita (payload + 4 byte di header), anch'essi in formato big-endian.
2. **Attesa della Conferma e Invio del Body**: Dopo aver spedito l'header, il server attende una risposta `"ok"` dal client. La ricezione di questa conferma segnala che il client è pronto a ricevere il payload, che viene quindi trasmesso. Questo meccanismo di handshake previene la perdita di dati e assicura una comunicazione ordinata.

La struttura `message` è il formato standard per tutti i dati scambiati. Essa include una sottostruttura `header` contenente la dimensione totale (`uint32_t size`) e una sottostruttura `body` che ospita i dati veri e propri come `std::vector<char> data`. Metodi come `read()` (per ottenere un `std::istringstream` dal body) e `size()` sono forniti per facilitare la manipolazione dei messaggi.

La chiusura della connessione, sia essa iniziata dal client o dovuta a errori, viene gestita attraverso `NetworkProvider::disconnect()` e il distruttore della classe, garantendo il rilascio delle risorse di rete. Il server, nel suo ciclo principale `Engine::loop()`, è progettato per gestire le connessioni dei client in modo sequenziale: terminata una sessione con un client (o a seguito di un errore, che viene loggato), il server torna in ascolto per nuove connessioni.

3.2. Python frontend

Il frontend è stato pensato per essere simile a una generica applicazione. Sviluppato in Python utilizzando il framework **PyQt5** per l'interfaccia grafica e diverse librerie specializzate per la gestione di mappe e geocoding, l'applicazione implementa un'architettura client-server che permette di calcolare percorsi alternativi tra due punti geografici utilizzando gli algoritmi precedentemente discussi nell'analisi del backend 3.1.

Descrizione dell'interfaccia:
Per la realizzazione della GUI abbiamo deciso di renderla semplice e intuitiva, simile ad applicazioni già note come Maps o Waze, con la caratteristica tuttavia di essere più personalizzabile la ricerca da parte dell'utente, vi sono infatti una serie di campi di input per l'utente da completare:

- partenza
- arrivo
- numero di percorsi da calcolare
- massima percentuale di sovrapposizione

Questo permette perciò all'utente non solo di scegliere quanti percorsi calcolare ma anche quanto diversi questi debbano essere tra loro.

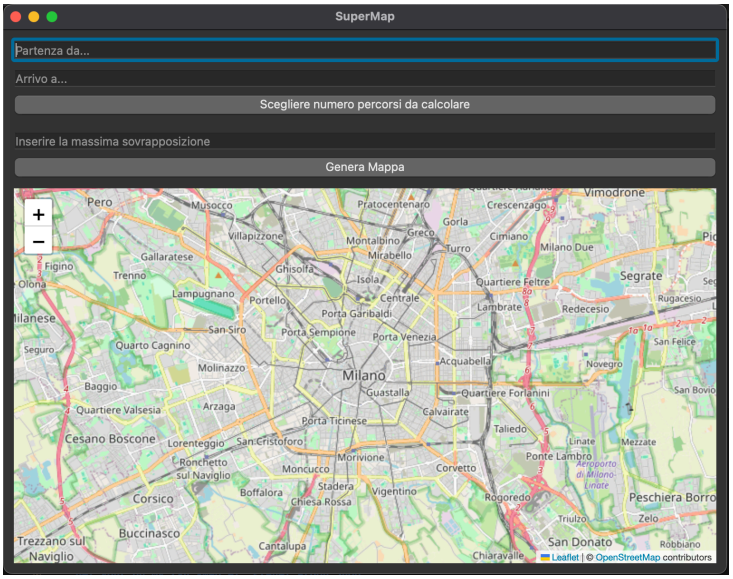

Figura 3.3. Interfaccia grafica

All'apertura dell'applicazione viene immediatamente visualizzata una mappa di default (nel nostro caso è stata scelta la mappa di Milano), che verrà aggiornata con i risultati non appena questi arriveranno dal backend in seguito al *click* dell'utente sul bottone **Genera Mappa**.

Tecnologie utilizzate
Per lo sviluppo del frontend si è reso necessario l'utilizzo di python client che potessero permetterci l'integrazione e la comunicazione con servizi come OSM o database di dati. In particolare sono stati utilizzati:

- **Geopy**: che è un python client usato per servizi di geocoding web
  - da geopy.geocoders viene importato **Nominatim** che, basandosi su OSM, converte i nomi di posti in coordiante geografiche
  - da geopy.distance invece viene utilizzato **Geodesic** per il calcolo preciso della distanza geografica tra due punti
- **Geonamescache**: database offline per informazioni demografiche. Il loro utilizzo è meglio specificato nella sezione 3.2.3

3.2.1. Client.py

La classe Client rappresenta il core dell'applicazione e gestisce l'intera interfaccia utente e la logica di comunicazione. Costruita come `class Client(QMainWindow)`, eredita da QMainWindow le funzioni necessarie per fornire una finestra principale completa con menu e barre degli strumenti.

Input Controls

Per gestire i parametri di input dell'utente vengono utilizzati:

- **QLineEdit**: utilizzato per leggere e registrare i campi di input di partenza, arrivo e sovrapposizione θ.
  Questi input vengono poi processati da un tool chiamato Nominatim che ha il compito di trasformare l'input dell'utente in un indirizzo specifico che possa essere riconducibile a un nodo in OSM:

```
geolocator=Nominatim(user\_agent="geoapp")
```

- **Dialog**: per specificare il parametro k (numero di percorsi alternativi)

Gestione della Comunicazione

La comunicazione con il server (backend), discussa lato-server nella sezione 3.1.2, nel frontend viene gestita tramite socket:

```
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client_socket.connect((host_server_ip, host_server_port))
```

Viene scelta una porta e un indirizzo ip su cui fare binding e su cui il server accetterà le connessioni.
Un aspetto molto importante dell'applicazione sviluppata sta nell'utilizzo di **QSocketNotifier** per gestire la ricezione asincrona dei risultati.
L'uso di questa classe si è reso necessario per stampare i risultati del backend man mano che vengono mandati e non tutti in una sola volta alla fine della computazione. Il funzionamento si basa su 2 eventi principali:

- **Monitoraggio Socket**: Rilevamento automatico di dati disponibili per la lettura

  ```
  QSocketNotifier(self.client_socket.fileno(), QSocketNotifier.Read)
  ```

  questo comando specifica che si vuole essere notificati non appena vi sono dati da leggere sul socket. `self.notifier` infatti chiamerà direttamente la funzione `receive_results` non appena verrà notificato di nuovi dati da leggere così da poter permettere la stampa del risultato

  ```
  self.notifier.activated.connect(self.receive_results)
  ```

- **Callback non bloccante**: una volta nella funzione `receive_results`, per gestire i risultati senza rischio di chiamate di callback per l'arrivo di nuovi dati, viene

temporaneamente disabilitato `self.notifier` con `self.notifier.setEnabled(False)` in modo da poter chiamare senza conflitti la funzione `results=receive_data(self.client_socket)` che stamperà su display i percorsi

Differenziazione Visuale dei Percorsi

Per la visualizzazione dei percorsi risultanti, si è scelto di utilizzare **marcatori rosa** per evidenziare i punti di **partenza** e di **arrivo**. I percorsi calcolati dai diversi algoritmi sono invece rappresentati mediante linee e marcatori dello stesso colore, assegnando un colore distinto a ciascun algoritmo:

- OnePass Plus: Linee rosse con marcatori rossi;
- ESX Algorithm: Linee blu con marcatori blu;
- Penalty Method: Linee verdi con marcatori verdi.

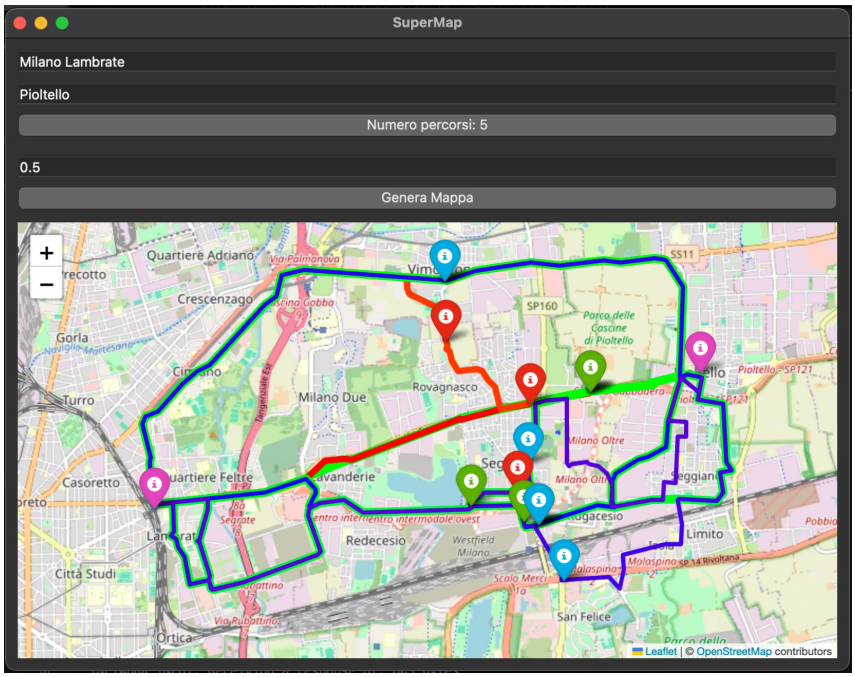

Figura 3.4. Stampa dei percorsi sulla mappa

Per distinguere ulteriormente i percorsi multipli generati dallo stesso algoritmo, è stato implementato un sistema di generazione dinamica di colori, che consente di variare

la sfumatura associata a ciascun percorso; dalla foto appena sopra si può facilmente notare questa differenza di sfumature per alcuni percorsi di ESX che sembrano essere più viola che blu.

3.2.2. graph_utils.py and network_utils.py

Graph Utils

Il modulo graph_utils fornisce funzionalità essenziali per la manipolazione dei grafi OpenStreetMap:

- **`add_osmid`**: ha lo scopo di aggiungere attributi OSMID ai nodi del grafo prima di essere mandati al backend. Viene utilizzata la libreria `lxml` e in particolare il modulo `etree` per il parsing e la manipolazione XML. Una volta eseguito il parsing del grafo in formato graphml, vi è la verifica della presenza di chiavi OSMID:
  - se esistono: viene assegnata la chiave unica al nodo
  - se non esistono: ne viene eseguita una creazione dinamica per poi assegnare l'ID univoco al nodo analizzato
- **`calc_min_dist_osmid`**: funzione che lavora sempre sul grafo e il cui obiettivo è calcolare i nodi più vicini alle coordiante di partenza e destinazione. Viene implementato un algoritmo di ricerca lineare tramite calcolo di distanza euclidea.

network_utils

Il modulo network_utils gestisce invece il protocollo di comunicazione client-server. Viene innazitutto definita la dimensione in byte del messaggio (meglio approfondita nella sezione 3.1):

```
msg_size_bytes = struct.pack("!i", graph_size, +8+8+4+4+4)
```

La `msg_size_bytes` permette al frontend di controllare che il messaggio arrivato sia della dimensione corretta prima di passare alla sua analisi interna tramite il **parsing**.

Parsing Risultati
I risultati che arrivano dal server vengono incapsulati nella calsse result

```
class Result:
    alg_name: str
    num_result: int
    length: float
    list_osmid: List[int]
```

cosi saranno facilmente decodificabili dalla classe client che dovrà stampare il risultato ricevuto

3.2.3. HierarchicalGraph.py

La classe HierarchicalGraph implementa un approccio innovativo alla gestione dei grafi stradali attraverso una struttura gerarchica multi-livello che prevede estrazioni più dettagliate in corrispondenza dei nodi di source e dest e una meno densa per la zona che collega i due nodi:

- Grafo Origine: Alto dettaglio nell'area di partenza
- Grafo Destinazione: Alto dettaglio nell'area di arrivo
- Grafo Collegamento: Bassa densità per i collegamenti a lunga distanza

La strategia di download si basa sull'utilizzo di **filtri stradali personalizzati** a seconda del grafo da estrarre.
Per i grafi stradali ad alta risoluzione viene infatti utilizzato un filtro che tiene conto di tutte le strade urbane più capillari:

```
custom_filter ='["highway"~"residential|tertiary|secondary|primary"]'
```

Al contrario, per l'estrazione del grafo centrale di collegamento vi è un filtraggio decisamente più elevato che porta ad avere come risultato finale solo strade principali

```
custom_filter = '["highway"~"trunk|motorway|primary"]'
```

Calcolo Dinamico del Raggio

Per l'estrazione delle zone si è deciso di utilizzare:

- **area circolare**: per le zone intorno a source e dest;
- **area rettangolare**: per la zona centrale.

É stato quindi sviluppato un algoritmo di calcolo dinamico del raggio basato sulla popolazione della città, in quanto ipotizzato una corrispondenza tra dimensione della città e numero di abitanti (ricavato tramite il servizio Geonamescache precedentemente nominato):

- Metropoli (>2M abitanti): 15km di raggio
- Grandi Città (1M-2M): 10km di raggio
- Città Medie (500K-1M): 7km di raggio
- Città Piccole (150K-500K): 5km di raggio
- Centri Minori (<150K): 2-4km di raggio

Per quanto riguarda invece il bounding-box di estrazione rettangolare abbiamo deciso di adottare un margine di circa 10km da latitudini e longituidini di partenza e arrivo.

3.2.4. Sistema di Caching

Quando l'utente inserisce gli input di partenza e destinazione, l'applicazione effettua due controlli:

- **self.G is None**: quindi controlla se l'utente ha deciso di utilizzare il grafo gia caricato in cahce
- **self.G.compare_trip(start, end)**: funzione di Hierarchical Graph che controlla se partenza e destinazione siano nelle stesse localita della computazione precedente

Questo permette un riuso intelligente dei grafi che vengono salvati con formato GraphML dopo ogni computazione in modo da velocizzare il processo di computazione.

4. HPC Test e Analisi dei dati

L'ultima fase è stata dedicata all'analisi di una serie di computazioni su vasta scala, effettuata utilizzando il sistema HPC CARLO del Politecnico di Milano. Questo stadio del progetto ha visto 3 principali momenti:

- scrittura di nuovo codice da utilizzare sul cluster
- sottomissione dei job di computazione
- analisi dei risultati ottenuti

4.1. Codice

L'esecuzione dell'intero processo di computazione è stata orchestrata tramite uno script bash (.sh), progettato per essere sottomesso come SLURM job sul sistema HPC.

Tecncologie utilizzate

Rispetto alla sezione 1.2, in questa fase si è reso necessario l'utilizzo anche di **Osmium Tool**, uno strumento utilizzabile da linea di comando progettato per manipolare file OpenStreetMap (OSM) in formati come .osm e .pbf.
Nel progetto è stato invocato per l'estrazione da bounding-box di aree e il loro conseguente filtraggio secondo le logiche descritte nella sezione 3.2.3.

4.1.1. Struttura dello scirpt

Lo script si occupa di iterare su una lista di coppie sorgente-destinazione, precedentemente compilata riguardante percorsi che spaziano tra Lombardia, Piemonte, Liguria e Val d'Aosta, contenute nel file `destinazioni.txt` con un k=7 fisso e un θ=0.5 . Per ciascuna riga del file, vengono eseguite le seguenti operazioni:

- **Setup iniziale**: attivazione dell'ambiente virtuale e controllo del file OSM richiesto.

- **Parsing e pulizia della riga corrente**, estraendo i **valori source e dest**, poi salvati come variabili d'ambiente.
- **Esecuzione di fileReader.py**, uno script Python che, come facevamo precedentemente, calcola le coordinate geografiche dei punti, determina i bounding box delle aree sorgente, destinazione e intermedia e infine salva queste informazioni in un file env_variables.txt
- **Lettura ed esportazione delle variabili d'ambiente** da env_variables.txt precedentemente riempito, per renderle accessibili anche agli strumenti successivi.
- **Estrazione delle sottoreti stradali** rilevanti tramite **Osmium**
- **Esecuzione di clientCarlo.py**, uno script simile a Client.py (discusso in 3.2.1) ma adattato all'ambiente batch del cluster, di conseguenza viene evitata la visualizzazione grafica (cioè il plot tramite folium di mappe e risultati) e si occupa di salvare i risultati dell'elaborazione direttamente su file CSV, ottimizzati per le successive analisi
- **Pulizia delle variabili d'ambiente e di env_variables.txt** al termine di ogni ciclo.

Dalla computazione di ogni riga vengono salvati su file :

- `Algorithm`: il nome dell'algoritmo che ha eseguito la computazione
- `Timestamp (s)`: il tempo impiegato per restituire quello specifico risultato. É importante sottolineare che il tempo riportato è di tipo **incrementale**, in quanto il calcolo del quinto percorso presuppone l'esecuzione sequenziale dei percorsi precedenti (dal primo al quarto), comportando quindi un timestamp necessariamente maggiore.
- `Relative number path`: che indica il numero del percorso relativo allo specifico algoritmo.
- `length (m)`: la lunghezza del percorso calcolato
- `source`: luogo di partenza
- `dest`: luogo di arrivo

4.1.2. Utilizzo del sistema HPC

Il cuore dell'esperienza è stato l'utilizzo di un sistema HPC reale che segue alcune regole ben precise.
Innanzitutto è stato necessario configurare correttamente le librerie di cui il nostro codice necessita, a tal fine l'intera toolchain è stata compilata da codice sorgente in modo da assicurare

la massima compatibilità con l'hardware installato nel sistema. In supporto a questa fase sono stati utilizzati i tool `spack` e `module`, il primo per trovare pacchetti dei vari moduli ed il secondo per renderli visibili nell'ambiente d'esecuzione. Fatto ciò è stato necessario costruire uno script shell che indicasse le modalità di esecuzione come ad esempio i percorsi agli eseguibili, le librerie richieste e le configurazioni del cluster stesso, come numero di nodi, tempo massimo di esecuzione e priorità. Sul sistema è infatti possibile sottomettere `job` di computazione gestiti da `Slurm Workload Manager` il quale si occupa di smistare le richieste di calcolo ai vari nodi allocando le risorse necessarie e gestendo l'esecuzione. Una volta sottomesso il nostro `job` la computazione è durata circa 20 ore producendo numerosi risultati che si sono rivelati di grande importanza nella fase di analisi.

4.1.3. Analisi dei risultati ottenuti

Infine ci siamo concentrati sull'analisi e approfondimento dei risultati ottenuti dai tre algoritmi di pathfinding (`onepass_plus, esx e penalty`) valutando le loro prestazioni attraverso diverse zone di estensione e utilizzando un sistema di **normalizzazione** basato su `oracoli` per garantire confronti equi.

fill_data.py

Innanzitutto, la prima cosa fatta è stata quella di *riempire* i dati laddove il backend non era riuscito a trovare alcun risultato entro il timeout imposto (5 min). Quindi vi è stata una prima computazione di `fill_data.py` sui dati del CSV in modo tale poter garantire un'analisi veritiera che tenesse conto anche del corretto yield.

dataAnalysis.py

Questo secondo script invece si premura di eseguire l'analisi vera e propria mediante diversi step:

Classificazione per estensione

Ogni percorso viene inizialmente classificato in base alla sua lunghezza, calcolata tramite un algoritmo di ricerca esatta (lenght_at_number_0), ad esempio Dijkstra o A*, a seconda di quello adottato da ciascun algoritmo. Questa lunghezza rappresenta il valore minimo possibile per il percorso dalla sorgente alla destinazione considerata:

- **Urban** (0-10 km): Percorsi urbani di breve distanza
- **ExtraUrban** (10-30 km): Percorsi extraurbani di media distanza
- **Regional** (30-90 km): Percorsi regionali di lunga distanza
- **InterRegional** (90+ km): Percorsi interregionali di lunghissima distanza

Successivamente, dopo aver fatto la ragionevole ipotesi che, dati un luogo di partenza e arrivo, è spesso sufficiente al cliente se vengono proposti almeno 3 percorsi, allora è stato filtrato il file eliminando tutte le righe che avessero `Relative number` che indicasse il quarto o quinto percorso calcolato (è da ricordare infatti che come detto nella sezione 4.1.1, per le computazioni si è scelto di usare k=7).

Approccio "Oracle-Based"

Il **concetto di oracle** si basa sul fatto che, per ogni coppia (origine, destinazione), lo script calcola un dato che rappresenta la performance teoricamente ottimale. In particolare, presi tutti i risultati calcolati da tutti e tre gli algoritmi per la specifica coppia (source, dest) analizzata, vengono estratti i migliori 3 tempi [1] e le rispettive lunghezze.
É importante specificare che i tempi per essere considerati tra i *migliori* devono essere inferiori al timeout (5min = 300s).
Gli oracoli estratti saranno dunque due:

- Oracle del tempo = media(tempi_migliori)
- Oracle della lunghezza = media(lunghezze_migliori_)

A questo punto perciò viene normalizzato ogni tempo e ogni lunghezza rispetto agli oracoli precedentemente calcolati tramite una semplice divisione:

$$\frac{dato\_da\_normalizzare}{oracle}$$

I numeri ottenuti risultano perciò tutti essere >=1 e sono indicatori di quanto un algoritmo si discosta dalla soluzione teoricamente ottimale.

Gestione dei casi limite

É importante fare un appunto su quelli che sono stati i casi limite riscontrati, in particolare, in alcuni momenti, nell'estrazione dei *gruppetti* (nel codice chiamati `range_data`) con relative_number_path uguale potevano verificarsi diversi scenari:

[1] Quando si parla di 3 tempi migliori ci si riferisce all'estrazione del miglior tempo tra i risultati ottenuti con `relative_number=1`, poi il migliore tra i risultati con `relative_number=2` e infine con `relative_number=3`

- **`len(range_data)=3`**: significa che tutti e tre gli algoritmi sono riusciti a trovare un percorso, si prende il minimo tra i 3
- **`0<len(range_data)<3`**: qualche algoritmo non ha trovato percorso, si salva il numero di algoritmi che non sono riusciti (per il calcolo successivo dello yield) e si prende il tempo migliore
- **`len(range_data)=0`**: nessun algoritmo ha trovato percorsi, quindi vengono selezionati come tempo e lunghezza migliore per quello specifico relative_number_path rispettivamente 300 e 0 (ovviamente sono aggiornati anche i dati per il calcolo dello yield)

Per il calcolo dell'oracle tuttavia vengono considerati solo i tempi validi (quindi <300) come detto precedentemente, di seguito un esempio per rendere il tutto più chiaro.

esempio: se per un percorso, vengono trovati solo uno o due risultati per ogni algoritmo, allora avremo i tempi e le lunghezze migliori come: best_times = [2.483469009399414, 2.54343910939237, 300] e best_lengths = [451.993, 473.765, 0]. In questo caso perciò per il calcolo degli oracoli di tempo e lunghezza verranno considerati solo i primi due risultati di ogni array e perciò avremo valid_times=[2.483469009399414, 2.54343910939237] e valid_lengths= [451.993, 473.765]

Altro caso limite riscontrato è stato invece quello relativo alla normalizzazione, in particolare infatti gli scenari possibili sono:

- l'algoritmo da normalizzare ha trovato un numero di percorsi pari al numero di tempi validi per il calcolo dell'oracolo, viene eseguita la normalizzazione con tutti i risultati negli array valid_*
- l'algoritmo da normalizzare ha trovato un numero di percorsi inferiore al numero di tempi validi, in questo caso viene **ricalcolato dinamicamente l'oracle** considerando solo un numero di risultati validi uguale al numero di percorsi trovati

É stata adottata questa scelta perchè in alcuni casi si rischiava di alzare troppo la media dell'oracolo facendo risultare l'algoritmo che non era riuscto a restituire sufficienti risultati come particolarmente veloce in quanto il risutlato della normalizzazione era inferiore a 1.

Livello 1 dell'analisi

Una volta terminata quindi l'estrazione degli oracoli e la successiva normalizzazione degli algoritmi. Si è passati, per ogni algoritmo e per ogni coppia (source, dest), a trovare i seguenti dati:

- **LM**: che rappresenta la lunghezza normalizzata media, calcolata quindi come media delle lunghezze medie normalizzate;
- **TM**: tempo normalizzato medio, calcolato con la stessa logica di LM;
- **avg_time**: tempo medio che ha impiegato l'algoritmo a trovare percorsi validi
- **avg_length**: lunghezza media dei percorsi trovati

Questi dati sono stati quindi incapsulati in delle entry da mettere all'interno di un dizionario:

```
entry = {
          'alg': alg,
          'source and dest': str(source)+","+str(dest),
          'LM': LM, #normalized Medium Lenght
          'TM': TM, #normalized Medium Time
          'avg_time_alg': avg_time,
          'avg_len_alg':avg_length,
          'avg_time_total': avg_best_time,
          'avg_length_total': avg_best_length,
          'category': category,
          'yield': 1-len(res_algs)/3
        }
```

Livello 2 dell'analisi

Una volta finita la fase 1 dell'analisi, si è passato a un'aggregazione dei dati precedentemente ottenuti. Quindi per ogni algoritmo e per ogni categoria geografica è stato calcolato il tempo e la lunghezza medi della categoria, il tempo e la lunghezza normalizzati medi della categoria e lo yield medio complessivo producendo quindi anche in questo caso delle entry del seguente tipo:

```
entry = {
          'category': category,
          'alg': alg,
          'avg_time_of_cat': alg_data_cat['avg_time_total'].mean(),
          'avg_length_of_cat':alg_data_cat['avg_length_total'].mean(),
          'avg_TM': alg_data_cat['TM'].mean(),
          'avg_LM': alg_data_cat['LM'].mean(),
          'avg_yield': alg_data_cat['yield'].mean(),
        }
```

Il risultato ottenuto è perciò un dizionario contenente 12 entry che rappresentano 4 categorie per tutti e 3 gli algoritmi:

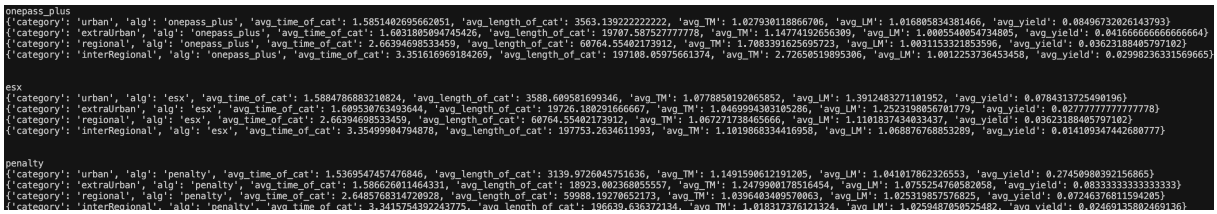

Figura 4.1. Risultati ottenuti

Da questa analisi perciò è stato possibile ricavare 4 grafi, uno per ogni categoria, in cui vi fossero 3 punti, uno per ogni algoritmo, che rappresentassero l'intersezione tra il tempo e la lunghezza medi impiegati dall'algoritmo nella restituzione di 3 percorsi.

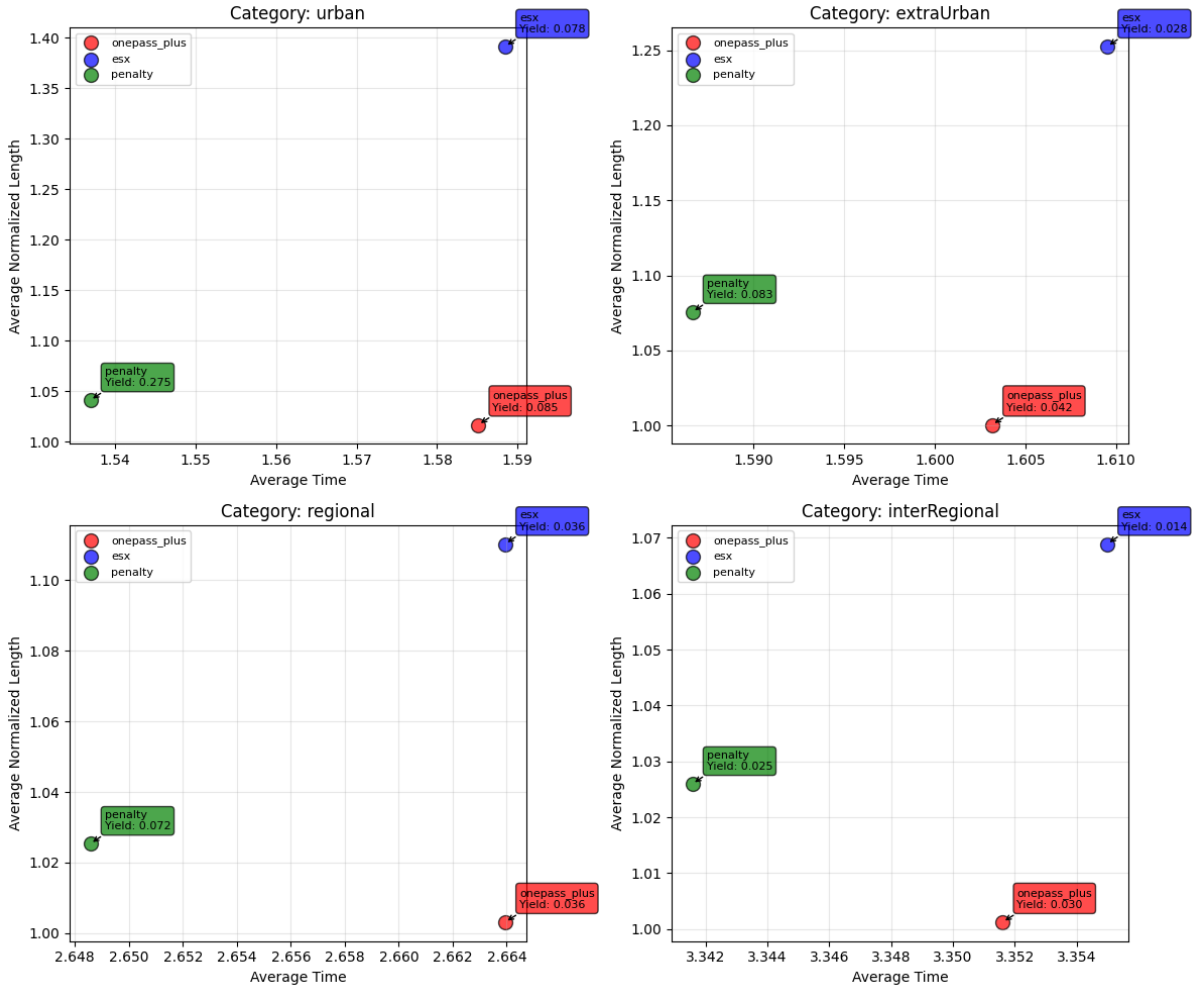

Figura 4.2. Grafici ottenuti

L'analisi applicata permette di identificare quale algoritmo sia più adatto per specifici contesti geografici, bilanciando velocità, qualità dei risultati e affidabilità computazionale.
In particolare, osservando i grafici generati, è possibile notare come ESX risulti più costoso sia in termini tempo che lunghezza, avendo tuttavia la caratteristica di essere il migliore per quanto riguarda lo yield.
Penalty sembra essere la scelta migliore avendo lunghezza media leggermente superiore a One-Pass+ ma tempo medio drasticamente inferiore, questo vantaggio è tuttavia *compensato* da uno yield molto alto rispetto agli altri due algoritmi, soprattutto in area urbana.
Possiamo perciò concludere che **tutti i metodi** possono essere considerati **"pareto-ottimali" nello spazio Yield-lunghezza-tempo**.

5. Conclusioni

Il progetto sviluppato in questi mesi si è rivelato estremamente **formativo**. Ci ha permesso di esplorare nuovi strumenti, librerie e tecnologie, offrendoci l'opportunità di comprenderli a fondo e metterli in pratica.

In particolare, l'ultima fase del progetto, dedicata ai test in ambiente HPC e all'analisi dei dati 4, ha rappresentato una novità assoluta rispetto agli argomenti trattati nei tre anni di studio. Questa parte ci ha consentito di confrontarci direttamente e in maniera concreta con sistemi di calcolo ad alte prestazioni, aprendoci a una nuova prospettiva sull'elaborazione e l'interpretazione di grandi moli di dati.

Abbiamo così potuto apprendere non solo l'utilizzo di questi strumenti avanzati, ma anche come l'analisi approfondita delle performance consenta di **valutare in modo oggettivo l'efficacia** di un algoritmo, aiutando a scegliere la soluzione più adatta a seconda del contesto applicativo.

Bibliografia

[1] C. Z. Andreas Paraskevopoulos, «Improved Alternative Route Planning,» *ATMOS - 13th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems*, pp. 108–122, 2013. DOI: 10.4230/OASIcs.ATMOS.2013.108. indirizzo: https://inria.hal.science/hal-00871739/PDF/10.pdf.

[2] T. Chondrogiannis, P. Bouros e J. Gamper, «Finding k-shortest paths with limited overlap,» *The VLDB Journal*, vol. 29, pp. 1023–1047, 2020. DOI: 10.1007/s00778-020-00604-x. indirizzo: https://doi.org/10.1007/s00778-020-00604-x.

[3] V. Akgün, E. Erkut e R. Batta, «On finding dissimilar paths,» *European Journal of Operational Research*, vol. 121, pp. 232–246, 2000. DOI: 10.1016/S0377-2217(99)00214-3. indirizzo: https://doi.org/10.1016/S0377-2217(99)00214-3.


POLITECNICO
MILANO