OraOra Rail: Sistema Avanzato di Visualizzazione e Planning dei mezzi di trasporto

Intelligent Systems - Automated Planning module

Ilaria Frandina
Matricola: 264232

Luglio 2025

Github project: `https://github.com/ila13-code/oraora-rail`

1 Introduzione

Questo report presenta **OraOra Rail**, un sistema completo per la visualizzazione e pianificazione automatica di itinerari di trasporto pubblico. Il progetto evolve dal semplice visualizzatore di orari ferroviari in un sistema per la pianificazione di viaggi utilizzando i dati di Trenitalia per la rete sarda di treni e autobus.

L'architettura del sistema comprende due componenti principali: un backend Flask che gestisce i dati ed effettua i calcoli di planning, e un frontend JavaScript per la visualizzazione interattiva. Il sistema supporta sia la visualizzazione di percorsi diretti (treni e autobus) che la pianificazione automatica di itinerari complessi. Tale pianificazione comprende cambi tra diversi mezzi di trasporto e permette l'ottimizzazione per tempo di viaggio o numero di cambi.

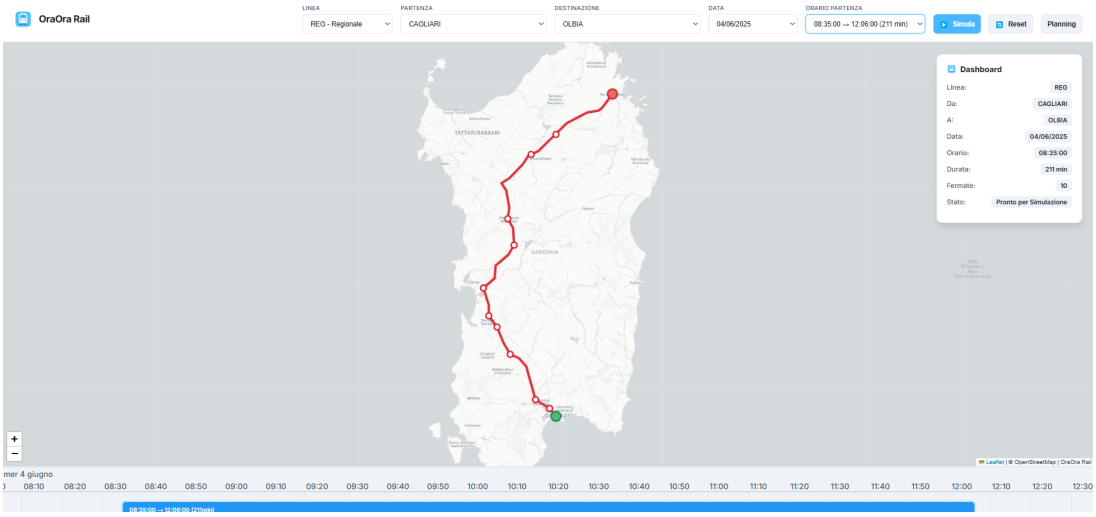

Figura 1: Interfaccia principale di OraOra Rail con controlli di selezione, mappa interattiva e dashboard informazioni

2 Architettura del Sistema e Stack Tecnologico

OraOra Rail è costruito su un'architettura a tre livelli che separa chiaramente preprocessing dei dati, logica di business e presentazione. L'architettura include un layer di planning che implementa algoritmi di ricerca per la risoluzione di problemi di routing nei trasporti.

2.1 Prepocessing dei dati

Il modulo **preprocess.py** rappresenta la fase fondamentale di trasformazione dei dati grezzi in strutture ottimizzate per la pianificazione automatica.

Il risultato finale del preprocessing è un insieme di file JSON ottimizzati: routes.json contiene le informazioni delle linee con modalità e colori, stops.json le fermate con coordinate geografiche, shapes.json i percorsi geografici, timetable.json tutti i viaggi con orari strutturati, e calendar.json le date che servono a determinare quando ogni servizio è attivo. Questa trasformazione riduce significativamente i tempi di caricamento e il parsing durante le operazioni di planning e di visualizzazione.

2.2 Backend Flask e Sistema di Planning

Il backend, implementato in Python con Flask, gestisce l'elaborazione e la manipolazione dei dati grezzi, e fornisce API REST per la pianificazione automatica e il recupero delle informazioni preprocessate.

```
@app.route("/plan", methods=["POST"])
def plan_endpoint():
    data = request.get_json(force=True)
    origin = data.get("origin")
    destination = data.get("destination")
    date = data.get("date")
    optimize = data.get("optimize", "time")  # 'time' o 'transfers'

    res = planner.plan(origin, destination, date, depart_after, optimize)
    return jsonify(res)
```

Listing 1: Servizio Rest di tipo POST messo a disposizione dal backend per il planning con tipo di ottimizzazione parametrica

2.2.1 Algoritmi di planning

Il sistema di pianificazione automatica contenuto in **planner.py** implementa due algoritmi, ciascuno ottimizzato per un criterio diverso: tempo minimo di viaggio e numero minimo di cambi.

L'idea di base è abbastanza semplice: il sistema considera ogni possibile spostamento da una fermata alla successiva come un "passo" che si può fare, tenendo conto degli orari dei mezzi. Per ogni fermata tiene traccia di quando ci si può arrivare e con quale mezzo, così da sapere se è possibile prendere una coincidenza successiva. Il cambio di mezzo avviene solo quando si passa da un viaggio (trip-id) a un altro diverso.

L'algoritmo **Earliest Arrival** implementa una ricerca che mantiene per ogni fermata il miglior tempo di arrivo raggiunto fino a quel momento. La struttura dati Label rappresenta lo stato di esplorazione e contiene il tempo di arrivo, il numero di cambi effettuati, l'ultimo viaggio utilizzato e i puntatori per ricostruire il percorso. Durante la scansione cronologica delle connessioni, l'algoritmo verifica se una nuova connessione può migliorare il tempo di arrivo a una fermata, considerando che deve esserci tempo sufficiente per effettuare eventualmente un cambio di veicolo. Il conteggio dei cambi avviene confrontando l'identificativo del viaggio corrente con quello precedente: se sono diversi si incrementa il contatore dei trasferimenti.

```
def _plan_earliest_arrival(self, conns: List[Connection], origin: str,
    destination: str, dep_after: int) -> Optional[Label]:
        best_arrival: Dict[str, Label] = {}
        best_arrival[origin] = Label(arr_time=dep_after, transfers=0, last_trip=
    None, prev=None, reached_by=None, stop=origin)

```

```
        for c in conns:
            if c.dep_time < dep_after:
                continue
            if c.dep_stop not in best_arrival:
                continue
            cur_label = best_arrival[c.dep_stop]
            if cur_label.arr_time <= c.dep_time:
                if (c.arr_time < best_arrival.get(c.arr_stop, Label(math.inf, 0, None, None, None, c.arr_stop)).arr_time):
                    new_label = Label(
                        arr_time=c.arr_time,
                        transfers=cur_label.transfers + (0 if cur_label.last_trip == c.trip_id else 1 if cur_label.last_trip is not None else 0),
                        last_trip=c.trip_id,
                        prev=cur_label,
                        reached_by=c,
                        stop=c.arr_stop
                    )
                    best_arrival[c.arr_stop] = new_label

        return best_arrival.get(destination)
```

Listing 2: Algoritmo Earliest Arrival

L'algoritmo **Minimum Transfers** funziona in modo diverso perché per ogni fermata conserva più soluzioni possibili invece di una sola. L'idea è semplice: una soluzione è migliore di un'altra se ha meno cambi e arriva prima (o almeno non peggio in entrambi gli aspetti). Questo permette al sistema di trovare itinerari che potrebbero essere più comodi per chi preferisce fare meno cambi anche se ci mette un po' di più. Per evitare che il numero di soluzioni cresca troppo e rallenti il sistema, l'algoritmo elimina costantemente quelle che sono chiaramente peggiori, mantenendo solo quelle che offrono un buon compromesso tra velocità e comodità.

```
def _plan_min_transfers(self, conns: List[Connection], origin: str, destination: str, dep_after: int) -> Optional[Label]:
        labels: Dict[str, List[Label]] = defaultdict(list)
        labels[origin].append(Label(dep_after, 0, None, None, None, origin))

        def dominated(new_l: Label, existing: List[Label]) -> bool:
            for l in existing:
                if l.transfers <= new_l.transfers and l.arr_time <= new_l.arr_time:
                    return True
            return False

        for c in conns:
            if c.dep_time < dep_after:
                continue

            for lab in list(labels[c.dep_stop]):
                if lab.arr_time <= c.dep_time:
                    transfers = lab.transfers + (0 if lab.last_trip == c.trip_id else 1 if lab.last_trip is not None else 0)
                    new_label = Label(
                        arr_time=c.arr_time,
                        transfers=transfers,
                        last_trip=c.trip_id,
                        prev=lab,
                        reached_by=c,
                        stop=c.arr_stop
                    )
                    if not dominated(new_label, labels[c.arr_stop]):
                        labels[c.arr_stop] = [l for l in labels[c.arr_stop] if not (new_label.transfers <= l.transfers and new_label.arr_time <= l.arr_time
```

```
    )]
                        labels[c.arr_stop].append(new_label)

        if not labels[destination]:
            return None

        return min(labels[destination], key=lambda l: (l.transfers, l.arr_time))
```

Listing 3: Algoritmo Minimum Transfers

Entrambi gli algoritmi condividono la fase di ricostruzione del percorso attraverso backtracking sui puntatori contenuti nei label. Il metodo **reconstruct** ripercorre la catena di label dalla destinazione all'origine, costruendo la sequenza di connessioni che formano l'itinerario ottimale. Il sistema gestisce naturalmente la multimodalità attraverso il campo mode delle connessioni, permettendo itinerari che combinano treni regionali e autobus.
L'output finale del planning è una struttura JSON completa che include: tempo totale, numero cambi, tipo di algoritmo utilizzato e la sequenza dettagliata di segmenti con orari e fermate. Questa informazione strutturata alimenta sia la visualizzazione su mappa che l'animazione del percorso, fornendo tutti i dettagli necessari per una rappresentazione completa dell'itinerario pianificato.

2.3 Frontend JavaScript Modulare

Il frontend è organizzato in moduli JavaScript che gestiscono separatamente:

- **DataManager**: Caricamento e gestione dati
- **MapManager**: Visualizzazione mappa interattiva con Leaflet
- **TimelineManager**: Timeline viaggi con Vis.js
- **AnimationManager**: Animazioni treni e autobus lungo i percorsi
- **UIManager**: Gestione interfaccia e stati applicazione
- **PlanningManager**: Interfaccia per il planning

3 Interfaccia Utente e Modalità Operative

Il sistema presenta due modalità operative distinte che si adattano a diversi scenari d'uso.

3.1 Modalità Visualizzazione Classica

Nella modalità classica è necessario inserire, in sequenza i seguenti dati: linea (BUS o REG), origine, destinazione, data, orario (vengono mostrati solo gli orari effettivamente presenti per la data selezionata) per la visualizzazione di percorsi diretti, come mostrato nella Figura 1.

3.2 Modalità Planning

La modalità planning introduce un pannello dedicato per la pianificazione di viaggi tra qualsiasi stazione presente:

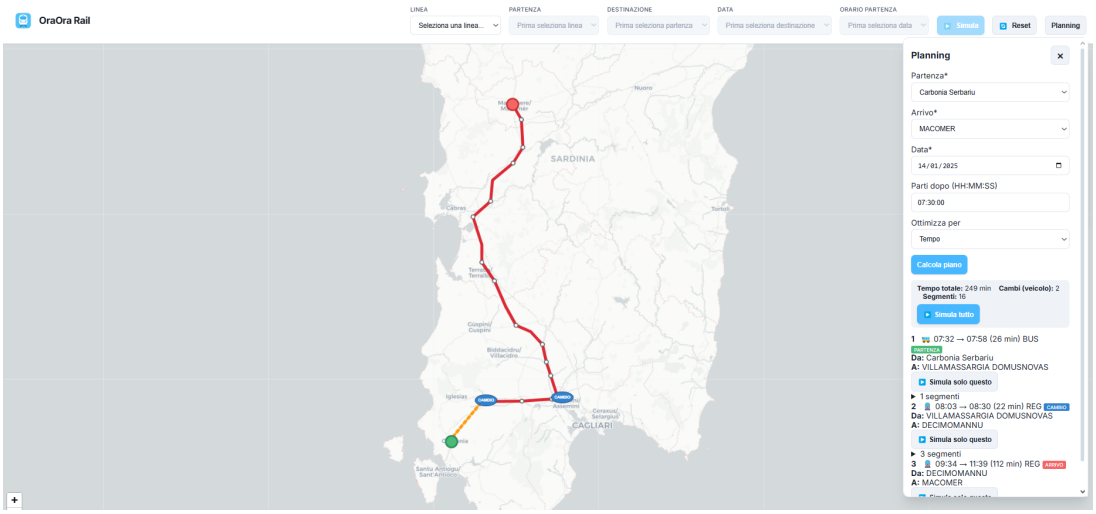

Figura 2: Interfaccia per itinerari con modalità Planning (in arancione la linea BUS - in rosso la linea TRENO)

Il pannello planning offre:

- Selezione libera di partenza e destinazione da tutte le stazioni disponibili
- Configurazione data e orario di partenza minimo
- Scelta del criterio di ottimizzazione (tempo vs. numero trasferimenti)
- Visualizzazione strutturata dei risultati con dettagli per ogni tratta
- Possibilità di simulare l'intero itinerario o singole tratte (come per la visualizzazione delle tratte dirette)

4 Risultati e Conclusioni

Il sistema gestisce con successo:

- Itinerari diretti su singole linee (treni e autobus)
- Itinerari con cambi tra treni e autobus
- Ottimizzazione per tempo minimo o numero minimo di trasferimenti
- Visualizzazione e animazione di percorsi

Il punto forte del progetto è che, caricando più dati nella cartella resources, prima di effettuare il preprocessing, il sistema si adatta e funziona su qualsiasi set di dati se questi hanno lo stesso formato. Questo rende tutto il sistema completamente scalabile.

Il sistema, inoltre, è containerizzato con Docker per semplificare deployment e gestione delle dipendenze. Nella cartella `docker` sono contenuti il Dockerfile e il file docker-compose per l'avvio completo del progetto.