POLITECNICO DI TORINO

Corso di Laurea Magistrale

Ingegneria Informatica (Computer Engineering)

Tesi di Laurea Magistrale

Servizi web innovativi per il trasporto merci intermodale: il progetto Synchro-NET

Relatore

Prof. Guido Perboli

Candidato

Giovanni Aquila

Luglio 2018

Sommario

Ringraziamenti 3

Introduzione 4

1 Sincromodalità e progetto Synchronet 6

1.1 Architettura del progetto Synchronet 6

1.2 SynchroNET Planner (SNETP) 9

1.3 OpenTripPlanner (OTP) 10

1.3.1 Data standard 11

1.4 Processo di ottimizzazione 11

1.5 Utilizzo del OTP 13

1.5.1 Esempio di utilizzo del OTP 17

2 Implementazione della “Capacity Mode” 26

2.1 “Capacity Mode”: che cos’è? 26

2.2 Specifiche della “Capacity Mode” 26

2.3 Framework Operativo 28

2.4 Analisi del codice implementato 29

2.4.1 CapacityCalculator 30

2.5 Conclusioni analisi codice 53

3 Analisi dei risultati 54

3.1 Esempio single destination mode 54

3.2 Esempio multi destination mode 59

3.3 Efficienza delle ricerche 65

4 Conclusioni 66

Bibliografia 67

Appendice 68

Ringraziamenti

Questa è la conclusione della mia carriera accademica e vorrei ringraziare prima di tutto la mia famiglia, mio padre e mia madre che hanno fatto di tutto per farmi arrivare fin qui e mia sorella Claudia che è stata sempre al mio fianco. Vorrei ringraziare tutti i compagni di corso che hanno condiviso con me questo percorso; i miei amici che mi hanno supportato psicologicamente, i colleghi che mi hanno ascoltato e incoraggiato durante le ore lavorative e i parenti che mi hanno aiutato da lontano. Vorrei ringraziare il Politecnico per l'opportunità di imparare, la Telematica Informatica per quella lavorativa, il Professor Perboli e il Professor Tadei per questa tesi e infine Riccardo per l'aiuto nel lavoro svolto in laboratorio nel realizzarla.

In particolare, vorrei ringraziare Alessio per il percorso svolto insieme, Roberto per l'amicizia da vent'anni, Eugenio per il supporto familiare e Gianluca per l'aiuto morale datomi in questi ultimi anni.

Infine, un ringraziamento a Carmela per la forza che mi ha trasmesso

Introduzione

SYNCHRO-NET (Synchro-modal Supply Chain Eco-Net)[1] è un progetto in corso, finanziato dal programma HORIZON 2020 della Commissione europea nell'ambito dell'accordo di sovvenzione n. 636354. L'**obiettivo** del progetto è dimostrare come un potente e innovativo "SYNCHRO-modal supply chain eco-NET" è in grado di catalizzare l'adozione del concetto di "slow steaming" e sincromodalità, garantendo solide **soluzioni economiche** che riducono la catena di fornitura per ridurre le emissioni e i costi per le operazioni logistiche e contemporaneamente aumentare l'affidabilità e i livelli di servizio per gli utenti della logistica. (Perboli et al., 2017) Il nucleo della soluzione SYNCHRO-NET sarà un eco-net di ottimizzazione e simulazione integrato, che incorpora: ottimizzazione della logistica sincrono-modale in tempo reale (abilitazione e-Freight), sistemi di simulazione e controllo della spedizione "slow steaming", rischio sincrono-modale / analisi dei benefici, modellazione statistica, soluzione dinamica per la valutazione dell'impatto delle parti interessate e un'architettura di comunicazione e governance sincronizzata. Forse l'elemento più importante di SYNCHRO-NET sarà la dimostrazione che lo "slow steaming", unita all'ottimizzazione della logistica sincrono-modale offre incredibili benefici a tutti gli attori della filiera: massiccia **riduzione delle emissioni** per il trasporto marittimo e terrestre a causa del trasferimento modale con modalità più verdi e processi di pianificazione ottimizzati che portano a **ridurre i chilometri** per i truck e **meno movimenti** di riposizionamento sprecati (Mes et al., 2016).

SynchroNET Planner (SNETP nel seguito) è la piattaforma software su cui si basa il progetto SYNCHRO-NET. Fondamentalmente, SNETP implementa le funzionalità di ottimizzazione SCD (Supply Chain De-stressing), insieme alle interfacce API per l'interazione con il modulo di ottimizzazione e analisi del rischio. Inoltre, il livello di presentazione di SNETP (l'interfaccia utente) fornirà il sistema di interazione di base per il Real-Time Synchro-Modal Optimisation. SNETP è strutturato come applicazione client-server. Il core di SNETP è costituito da 3 componenti di base:

- un "**graph builder**": elabora dati di input geospaziali e di transito e crea un grafico "logico" e più generale su cui sono possibili analisi quantitative;
- un "**routing engine**": in grado di trovare uno (o più) percorso possibile percorribile da un nodo ad un altro nodo nel grafico e di valutarlo in termini di parametri quantitativi;

[1] SYNCRO-NET sito web: *https://www.synchronet.eu* (accesso il 12/06/18)

- un “**user interface**”: offre all'utente la possibilità di interagire con il planner, al fine di trovare la soluzione migliore per le sue esigenze.

Lo sviluppo della piattaforma SNETP avviene su una raccolta di librerie open-source esistente chiamata OpenTripPlanner (OTP)[2]. Questo pacchetto, scritto in Java[3], segue un'**architettura client-server** e fornisce un'interfaccia Web e API REST per l'utilizzo da parte di applicazioni di terze parti.

Questa **tesi** si concentra sull’implementazione della “**Capacity Mode**”: il lavoro svolto riguarda la modifica del software esistente con la possibilità dell’utente di includere nel processo di simulazione anche un numero di container da spedire alle varie destinazioni scelte.

Nello specifico, la tesi è strutturata come segue:

- Il primo capitolo illustra nel dettaglio il progetto Synchronet come già anticipato in questa introduzione;
- Il secondo capitolo illustra nel dettaglio l’implementazione della “Capacity Mode”;
- Il terzo capitolo conclude questa tesi con l’output generato dall’OTP con la “Capacity Mode” attiva.

[2] Open Trip Planner sito web: *http://www.opentripplanner.org* (accesso il 12/06/18)
[3] Java informazioni: *https://www.java.com/it/download/faq/whatis_java.xml* (accesso il 12/06/18)

1 Sincromodalità e progetto Synchronet

In questo capitolo verrà spiegato nel dettaglio il progetto Synchronet e la sua struttura come già preannunciato nell'introduzione.

1.1 Architettura del progetto Synchronet

SYNCHRO-NET si affida alla tecnologia orientata ai servizi e al flusso di lavoro per implementare una piattaforma integrale. Questa piattaforma consente alle reti logistiche di comunicare con la piattaforma secondo necessità in modo estremamente dettagliato e di interagire facilmente con gli altri soggetti della logistica. La piattaforma utilizza la **cooperazione** e il **coordinamento** di diversi moduli logistici, consentendo l'integrazione con servizi esterni utilizzando API standardizzate e aperte. L'obiettivo è sviluppare o migliorare tutti gli strumenti di e-Freight e i meccanismi di coordinamento intelligente che consentano di pianificare e ottimizzare in modo efficace una complessa catena di distribuzione sincrono-modale, sia a livello strategico che in tempo reale. La Figura 1 mostra un'architettura semplificata dei moduli logistici coinvolti e delle loro connessioni.

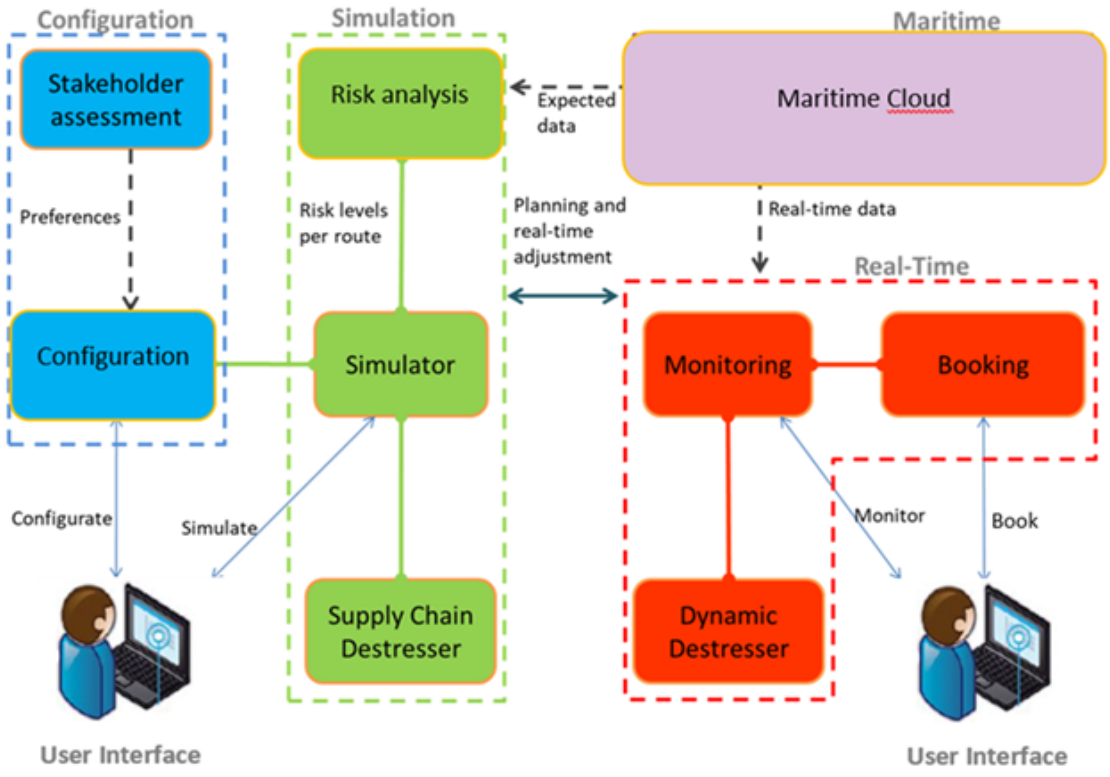


Figura 1 - Interazione semplificata di architettura e moduli logistici

Più tecnicamente, il processo di ottimizzazione coinvolge tre moduli principali che lavorano in modo cooperativo: "Real-Time Synchro-Modal Optimisation" (RTO), "dynamic Supply Chain De-stressing" (SCD) e "Risk analysis and optimization" (RAO) (Holfeld et al., 2018a; Simroth et al., 2016). La Figura 2 illustra le principali interazioni tra i tre moduli di ottimizzazione.

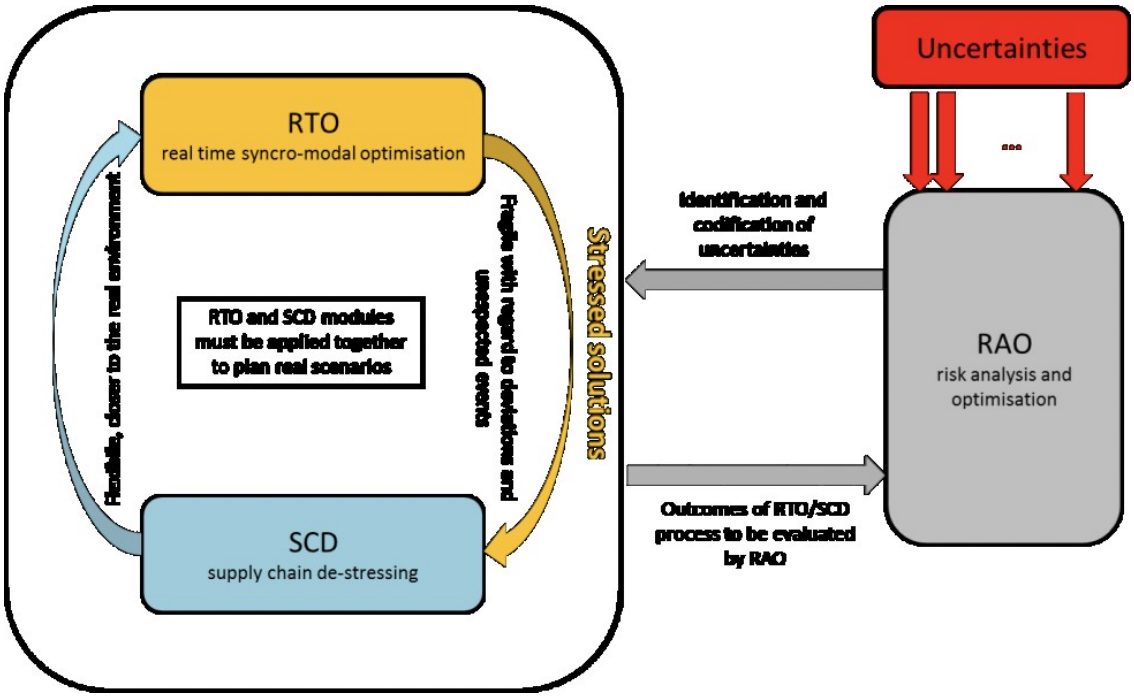


Figura 2 - Interazione principale tra i moduli SCD, RAO e RTO

L'RTO definisce soluzioni sollecitate per l'intero sistema e le passa alla SCD. La SCD collabora quindi con la RAO per ridurre il carico di soluzioni utilizzando diversi obiettivi basati sulla valutazione degli indicatori chiave di prestazione (KPI) e degli indicatori chiave di rischio (KRI) (Holfeld et al., 2018b). Alcuni esempi sono il tempo di consegna minimo, il costo totale minimo, l'emissione totale minima, il trade-off tra il rischio di ritardo e il risparmio dei costi quando vengono prese in considerazione le informazioni in tempo reale e gli eventi imprevisti. L'obiettivo del modulo SCD è di de-stress le soluzioni di pianificazione trasporto sincro-modali cooperando con RTO e RAO. Il modulo SCD è costituito da due componenti:

1. De-stresser database e motore di ricerca (database che memorizza informazioni su decisioni storiche, ad esempio rotte, modalità, società, in modo che possa essere utilizzato come riferimento per richieste future da parte degli utenti);
2. De-stresser della supply chain (fornire soluzioni de-stressed, tenendo conto del KPI e valutazione del rischio).

La Figura 3 mostra il flusso di dati tra i moduli.

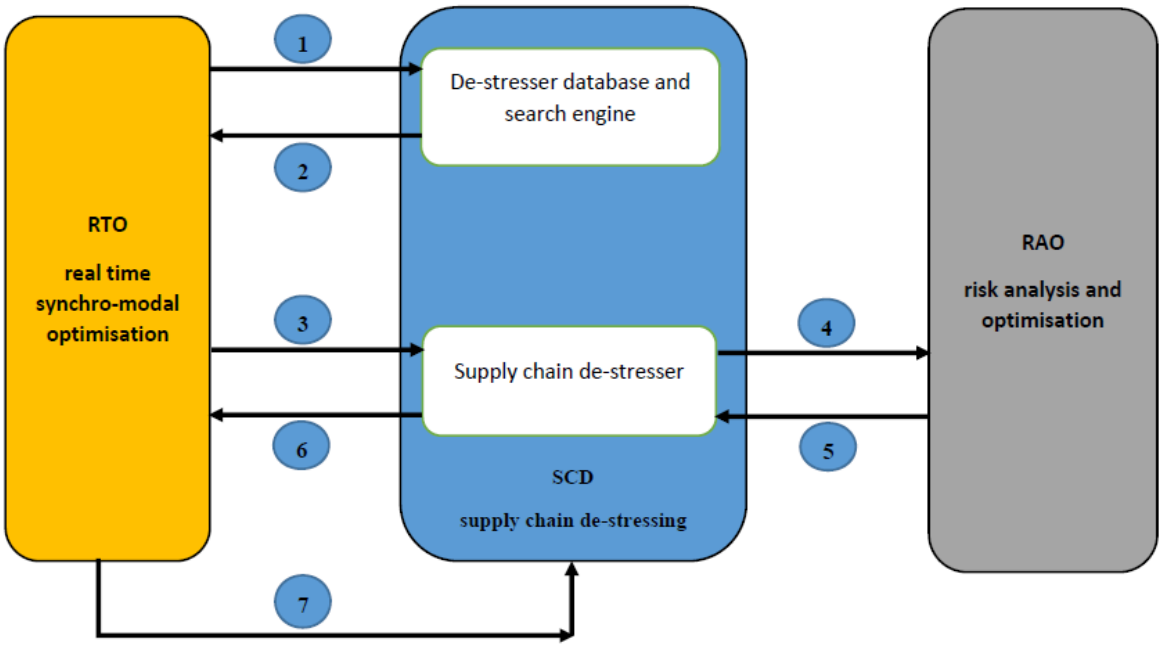


Figura 3 - Flusso di dati tra moduli SCD, RTO e RO

Più precisamente:

- L'utente crea una richiesta al sistema (ad es., Inviando merci dalla posizione A alla posizione E e altre specifiche come le preferenze dell'utente) e l'RTO invia questa richiesta al componente "De-stresser database and search engine".
- Il componente "De-stresser database and search engine" risponde con un elenco di possibili percorsi A-B-C-D-E. Per ciascun collegamento di ciascuna rotta, includerà il trasportatore / servizio effettivo utilizzato in precedenza.
- L'RTO elabora questi n percorsi e li riduce a m percorsi che sono fattibili, tenendo conto dei tempi / orari di servizio effettivi. Quindi passa questi m percorsi (compresi i dettagli sui tempi di ciascuna gamba) al componente "Supply chain de-stresser".
- Il componente "supply chain de-stresser" calcola i "punteggi" per ciascun m route in base agli indicatori chiave di prestazione (KPI) (ad esempio, il minimo del tempo di consegna, il minimo del costo totale, il minimo dell'emissione totale). Li invia anche alla RAO per l'analisi dei rischi.
- Il RAO determina gli indicatori di rischio chiave (KRI) (ad esempio, deviazione temporale attesa (numero di ore aggiuntive), deviazione dei costi attesi, probabilità di danno / perdita (%), tasso di flessibilità) per ciascun percorso rilevante fornito dalla "supply chain de-stresser". Ogni rotta viene valutata in base al suo

comportamento riguardo a varie interruzioni durante l'esecuzione di un percorso. Il RAO quindi passerà i KRI per ciascuna delle rotte rilevanti.

- Il componente "supply chain de-stresser " invia i "punteggi" definiti da KPI e KRI per ciascun percorso pertinente fino all'RTO.
- Il sistema (RTO) mostra un elenco di possibili percorsi multimodali, insieme a KPI e KRI (costo, rischio, ambiente) e l'utente decide sulla rotta preferita. Successivamente confermeranno la compagnia di trasportatori o il servizio ferroviario o la compagnia / viaggio da utilizzare e le tariffe concordate. L'RTO passerà queste informazioni riservate al modulo SCD in modo che possa essere utilizzato per suggerimenti futuri a questo utente, rendendo più veloce la procedura di preventivi e prenotazioni.

1.2 SynchroNET Planner (SNETP)

Come già anticipato nell'introduzione, SynchroNET Planner è la piattaforma software su cui si basa il progetto SYNCHRO-NET e il suo core è costituito da 3 componenti di base: graph builder, routing engine e user interface (figura 4).

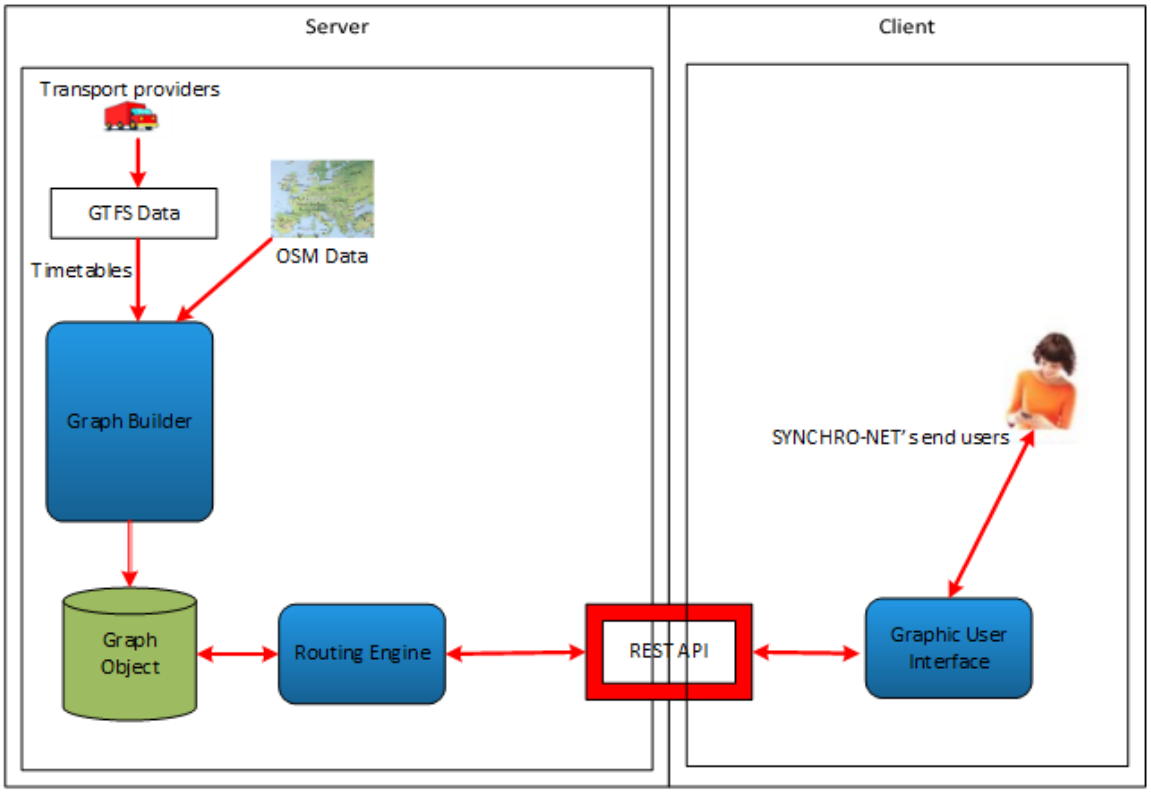


Figura 4 – Componenti del SNETP

Più precisamente, Graph Builder è un'applicazione Java che unisce i dati OSM con i dati GTFS forniti dal sistema di trasporto (vedere la sezione successiva per i dettagli) e genera un **oggetto grafico**. Questo oggetto grafico è un file di testo basato su **XML** (eXtensible Markup Language)[4] e viene creato una volta configurato il server. Graph Object è l'oggetto dati principale su cui operano tutti gli algoritmi di pianificazione del percorso implementati nel motore di routing. Il motore di routing è responsabile dell'elaborazione delle richieste dall'interfaccia utente grafica (GUI), estraendo i dati richiesti dall'oggetto grafico, creando la risposta e inviandola alla GUI. La GUI è un'**interfaccia web** (vedere la versione corrente di esso in Figura 5).

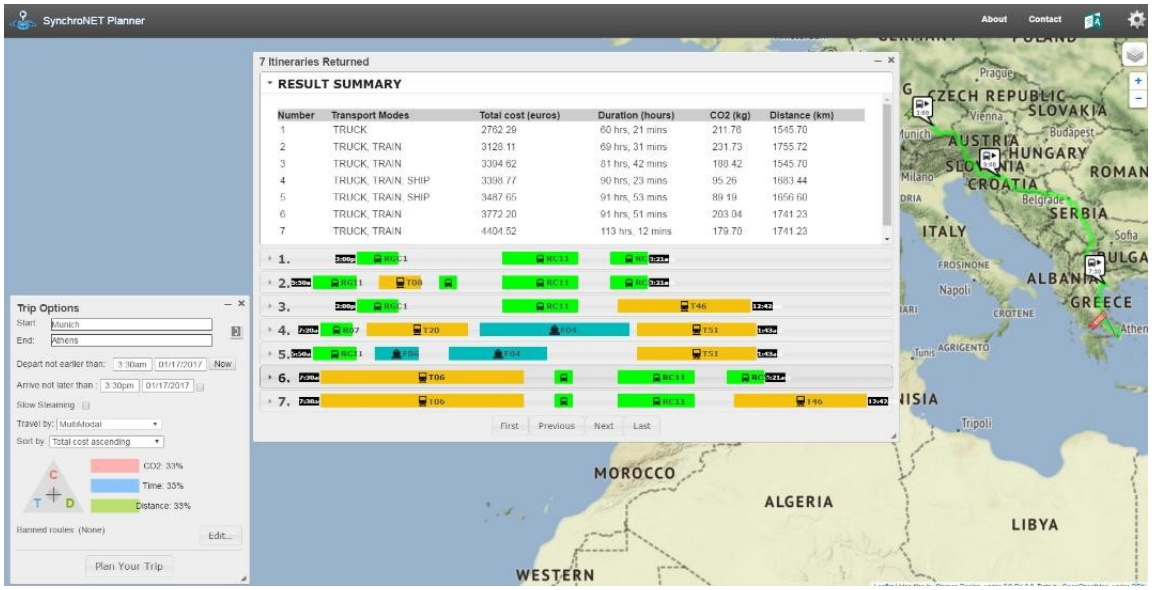


Figura 5 – Interfaccia utente

1.3 OpenTripPlanner (OTP)

Come già trattato nell'introduzione OTP è un package scritto in Java che fornisce un'interfaccia web con un'architettura client-server per l'utilizzo del simulatore (Giusti et al., 2018).

[4] XML sito web: *https://www.w3.org/XML/* (accesso il 12/06/18)

1.3.1 Data standard

OTP funziona principalmente su due standard di dati aperti, ovvero General Transit Feed Specification (GTFS)[5] per i transiti e Open Street Map (OSM)[6] per le reti stradali.

1.3.1.1 Open Street Map (OSM)

Open Street Map (OSM) è un progetto di mappatura collaborativa a livello mondiale che mira a creare una mappa modificabile gratuita del mondo intero in modalità condivisa. OTP legge i dati OSM (da uno o più file) come input per creare il grafico / la rete su cui sono costruiti i percorsi.

1.3.1.2 General Transit Feed Specification (GTFS)

Il General Transit Feed Specification (GTFS) viene utilizzato per generare il programma per un elenco di rotte, in cui ogni rotta è definita da alcune caratteristiche quali origine, destinazione, modalità di trasporto, ora di partenza e orario di arrivo ad ogni fermata che passa attraverso il percorso, etc...

1.4 Processo di ottimizzazione

Il processo di ottimizzazione si basa su degli algorimti di ricerca per i cammini minimi come ad esempio l'algoritmo di Dijkstra.

> *"Supponiamo di avere un grafo con n vertici contraddistinti da numeri interi {1,2,...,n} e che uno di questi nodi sia quello di partenza e un altro quello di destinazione. Il peso sull'arco che congiunge i nodi j e k è indicato con p(j,k). A ogni nodo, al termine dell'analisi, devono essere associate due etichette, f(i) che indica il peso totale del cammino (la somma dei pesi sugli archi percorsi per arrivare al nodo i-esimo) e J(i) che indica il nodo che precede i nel cammino minimo. Inoltre definiamo due insiemi S e T che contengono rispettivamente i nodi a cui sono già state assegnate le etichette e quelli ancora da scandire.*

[5] GTFS sito web: *http://developers.google.com/transit/gtfs/* (accesso il 12/06/18).
[6] OpenStreetMap sito web: *http://www.openstreetmap.org* (accesso il 12/06/18).

1. *Inizializzazione.*
   Poniamo S={1}, T={2,3,...,n}, f(1)=0, J(1)=0.
   Poniamo f(i)=p(1,i), J(i)=1 per tutti i nodi adiacenti ad 1.
   Poniamo f(i)= ∞, per tutti gli altri nodi.

2. *Assegnazione etichetta permanente*
   Se f(i)= ∞ per ogni i in T ***STOP***
   Troviamo j in T tale che f(j)=min f(i) con i appartenente a T
   Poniamo T=T{j} e S=S∪{j}
   Se T=Ø ***STOP***

3. *Assegnazione etichetta provvisoria*
   Per ogni i in T, adiacente a j e tale che f(i)>f(j)+p(j,i) poniamo:
   f(i)=f(j)+p(j,i)
   J(i)=j
   Andiamo al passo 2"[7]

Ma dato che l'OTP è un sistema complesso di simulazione, bisogna considerare che il processo di ottimizzazione riguarda una rete multi-modale con un numero elevato di nodi presenti (città); inoltre all'interno del processo bisogna considerare le rotte come archi del grafo orientato con il loro peso; in più ci sono vincoli addizionali che l'utente può inserire come la forzatura di una certa rotta, oppure città da evitare. Ulteriormente bisogna considerare che: esistono settaggi input sulla tipologia di trasporto (solo nave, treno o truck); si può scegliere se dare più importanza alla distanza totale o alle emissioni; si può dare come priorità la durata totale dell'itinerario o aggiungere pesi e misure su altre tipologie di settaggi (evita un certo tipo di strade, costi vari orari o per km, analisi del rischio…).

[7] Algoritmo di Dijkstra Wikipedia: https://it.wikipedia.org/wiki/Algoritmo_di_Dijkstra (accesso il 29/06/2018)

1.5 Utilizzo del OTP

Come già spiegato precedentemente il simulatore viene utilizzato con un'interfaccia web (Figura 6) con architettura client-server dove dati dei dati di input il server risponde con un file XML che l'interfaccia rielabora e mostra all'utente.

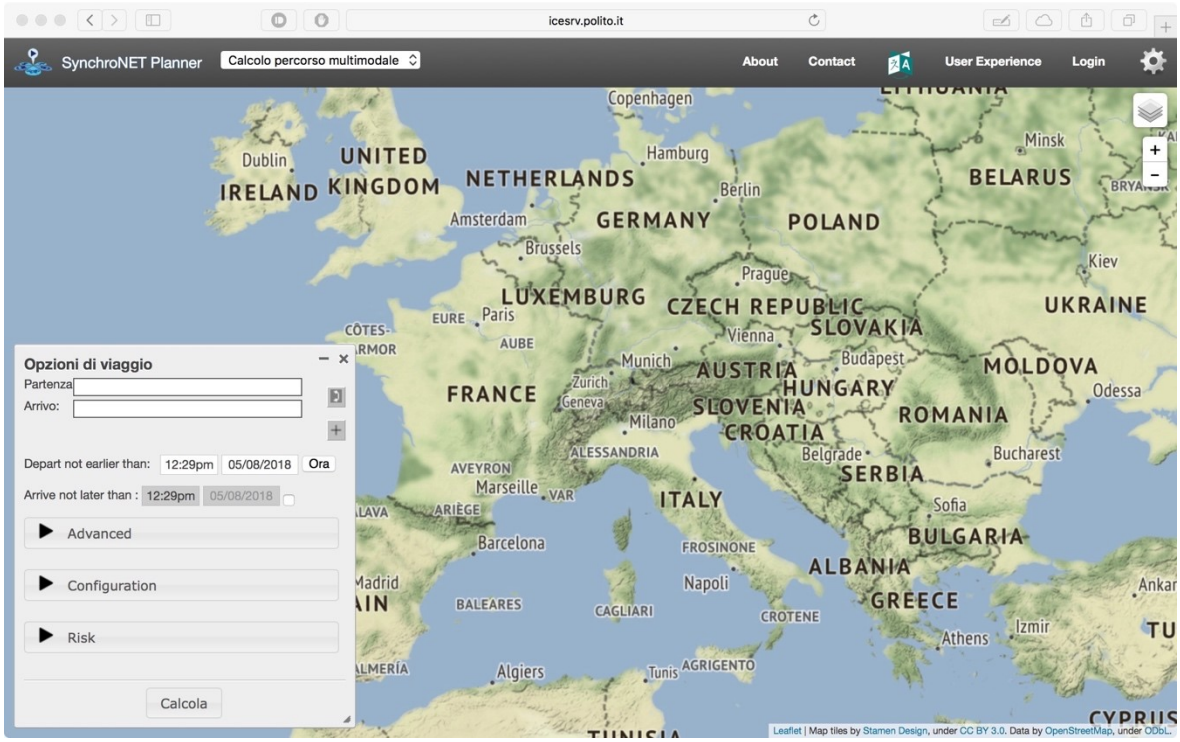


Figura 6 – Home page

Nella figura 7 (a, b, c) possiamo vedere il formato dell'XML che il server manda al client che poi l'interfaccia grafica elabora e mostra all'utente:

```
▼<Response>
  ▼<requestParameters>
      <date>05-16-2018</date>
      <riskAnalysis>false</riskAnalysis>
      <wheelchair>false</wheelchair>
      <co2_per_km_train>0.06</co2_per_km_train>
      <fromPlace>Cork::51.898,-8.47061</fromPlace>
      <cost_distance_per_km_truck>0.4</cost_distance_per_km_truck>
      <alighting_truck>3600</alighting_truck>
      <alighting_ship>10800</alighting_ship>
      <capacity_ship>1000</capacity_ship>
      <locale>en</locale>
      <mode>TRANSIT,WALK</mode>
      <cost_per_hour>30</cost_per_hour>
      <arriveBy>false</arriveBy>
      <capacity_train>200</capacity_train>
      <distance_w>0.334</distance_w>
      <co2_w>0.333</co2_w>
      <toPlace>Patra::38.2444,21.7344</toPlace>
      <co2_per_km_ship>80</co2_per_km_ship>
      <slowSteaming>false</slowSteaming>
      <boarding_ship>10800</boarding_ship>
      <co2_cost_per_kg>0.15</co2_cost_per_kg>
      <cost_distance_per_km_train>0.0324</cost_distance_per_km_train>
      <sortResultType>0</sortResultType>
      <co2_per_km_truck>0.25</co2_per_km_truck>
      <KRI_time_w>0.25</KRI_time_w>
      <KPIimportance>false</KPIimportance>
      <time_w>0.333</time_w>
      <co2_per_km_ship_Slow>60</co2_per_km_ship_Slow>
      <maxWalkDistance>804.672</maxWalkDistance>
      <boarding_truck>3600</boarding_truck>
      <alighting_ship_roro>7200</alighting_ship_roro>
      <KRI_flexibility_w>0.25</KRI_flexibility_w>
      <boarding_train>7200</boarding_train>
      <alighting_train>7200</alighting_train>
      <timeArriveVisible>false</timeArriveVisible>
      <boarding_ship_roro>7200</boarding_ship_roro>
      <KRI_cost_w>0.25</KRI_cost_w>
      <truck_speed>0</truck_speed>
      <co2_per_km_ship_Fast>100</co2_per_km_ship_Fast>
      <cost_distance_per_km_ship>0.108</cost_distance_per_km_ship>
      <time>8:00pm</time>
      <capacityRequest>500</capacityRequest>
      <KRI_safety_w>0.25</KRI_safety_w>
    </requestParameters>
  ▼<plan>
      <date>1526493600000</date>
    ▼<from>
```


Figura 7a – XML rielaborato successivamente dall'interfaccia grafica

In particolare, nella 7a troviamo l'**input** che il client ha mandato al server con i nodi di partenza e arrivo, date, orari, numero di containers e tutte le opzioni che l'utente può selezionare nell'interfaccia utente.

```
▼<itineraries>
  ▼<itineraries>
    ▶<verticesLabels>...</verticesLabels>
     <operationalTime>0</operationalTime>
     <travelTime>0</travelTime>
     <waitingTime>145079</waitingTime>
     <costDistance>0.0</costDistance>
     <costCO2>29876.73</costCO2>
     <costTime>0.0</costTime>
     <travelTimeCost>0.0</travelTimeCost>
     <operationalCost>0.0</operationalCost>
     <waitingCost>0.0</waitingCost>
     <travelCost>0.0</travelCost>
     <duration>481852</duration>
     <fromDatabase>false</fromDatabase>
     <elapsedTime>510652</elapsedTime>
     <startTime>1526493600000</startTime>
     <endTime>1527004252000</endTime>
     <distance>3569.0</distance>
     <distanceSHIP>1981000.0</distanceSHIP>
     <distanceTRUCK>0.0</distanceTRUCK>
     <distanceTRAIN>1588000.0</distanceTRAIN>
     <CO2>199178.2</CO2>
     <CO2SHIP>79240.0</CO2SHIP>
     <CO2TRUCK>119700.0</CO2TRUCK>
     <CO2TRAIN>238.20000000000073</CO2TRAIN>
     <totalcost>34170.7042</totalcost>
     <totalcostSHIP>13391.898000000001</totalcostSHIP>
     <totalcostTRUCK>19897.133333333335</totalcostTRUCK>
     <totalcostTRAIN>881.6728666666662</totalcostTRAIN>
     <haveTRUCK>false</haveTRUCK>
     <haveSHIP>true</haveSHIP>
     <haveTRAIN>true</haveTRAIN>
     <quality>100</quality>
     <id>65</id>
    ▶<KRIs>...</KRIs>
     <fare/>
    ▼<legs>
      ▼<legs>
          <destinationId>1</destinationId>
          <previousDestinationId>0</previousDestinationId>
          <finalDestination>0</finalDestination>
          <boarding>21600</boarding>
          <alighting>0</alighting>
          <transit>0</transit>
          <capacity>1000</capacity>
          <assignedcontainers>500</assignedcontainers>
          <costforonecontainerTruck/>
          <capacityId>1</capacityId>
          <prevCapacityId/>
          <nextCapacityId/>
```


Figura 7b – XML riguardante l'itinerario

Nella figura 7b invece si trovano tutte le informazioni riguardante l'**itinerario**, in questo caso il primo della lista con poi le stesse informazioni che si trovano dopo nell'output

dell'interfaccia grafica. Infine, nella 7c troviamo le informazioni riguardo alla **singola tratta**.


```
▼<legs>
  ▼<legs>
      <destinationId>1</destinationId>
      <previousDestinationId>0</previousDestinationId>
      <finalDestination>0</finalDestination>
      <boarding>21600</boarding>
      <alighting>0</alighting>
      <transit>0</transit>
      <capacity>1000</capacity>
      <assignedcontainers>500</assignedcontainers>
      <costforonecontainerTruck/>
      <capacityId>1</capacityId>
      <prevCapacityId/>
      <nextCapacityId/>
      <startTime>1526544000000</startTime>
      <endTime>1526605982000</endTime>
      <startTimeDisturbed>0</startTimeDisturbed>
      <endTimeDisturbed>0</endTimeDisturbed>
      <id>180</id>
      <departureDelay>0</departureDelay>
      <arrivalDelay>0</arrivalDelay>
      <realTime>false</realTime>
      <isNonExactFrequency/>
      <headway/>
      <distance>792000.0</distance>
      <costdistance>85.536</costdistance>
      <costtime>516.5166666666667</costtime>
      <CO2>31680.0</CO2>
      <costCO2>4752.0</costCO2>
      <totalcost_multiobjective>1842.925074</totalcost_multiobjective>
      <totalcost>5354.0526666666665</totalcost>
      <pathway>false</pathway>
      <mode>FERRY</mode>
      <route>Medium</route>
      <agencyName>Fake Ships</agencyName>
      <agencyUrl>https://www.fakeships.com/</agencyUrl>
      <agencyTimeZoneOffset>7200000</agencyTimeZoneOffset>
      <routeColor/>
      <routeType>4</routeType>
    ▶<routeId>...</routeId>
      <routeTextColor/>
      <interlineWithPreviousLeg>false</interlineWithPreviousLeg>
      <tripShortName/>
      <tripBlockId/>
      <headsign/>
      <agencyId>FS</agencyId>
    ▶<tripId>...</tripId>
      <serviceDate>20180517</serviceDate>
    ▶<from>...</from>
    ▶<to>...</to>
```


Figura 7c – XML riguardante la singola tratta

1.5.1 Esempio di utilizzo del OTP

Ora vorrei illustrare il concreto funzionamento del simulatore senza l'implementazione che verrà illustrata nel secondo capitolo. Il simulatore funziona sia con una sola destinazione che con destinazioni multiple.

1.5.1.1 Single destination mode

Questo è un esempio di modalità "**single destination mode**".
Nella figura 8 abbiamo inserito come partenza la città di **Cork** (Irlanda) e come arrivo la città di **Patrasso** (Grecia; Patra in lingua originale). Dopodiché si può impostare sia la data e l'ora di partenza che quella di arrivo.

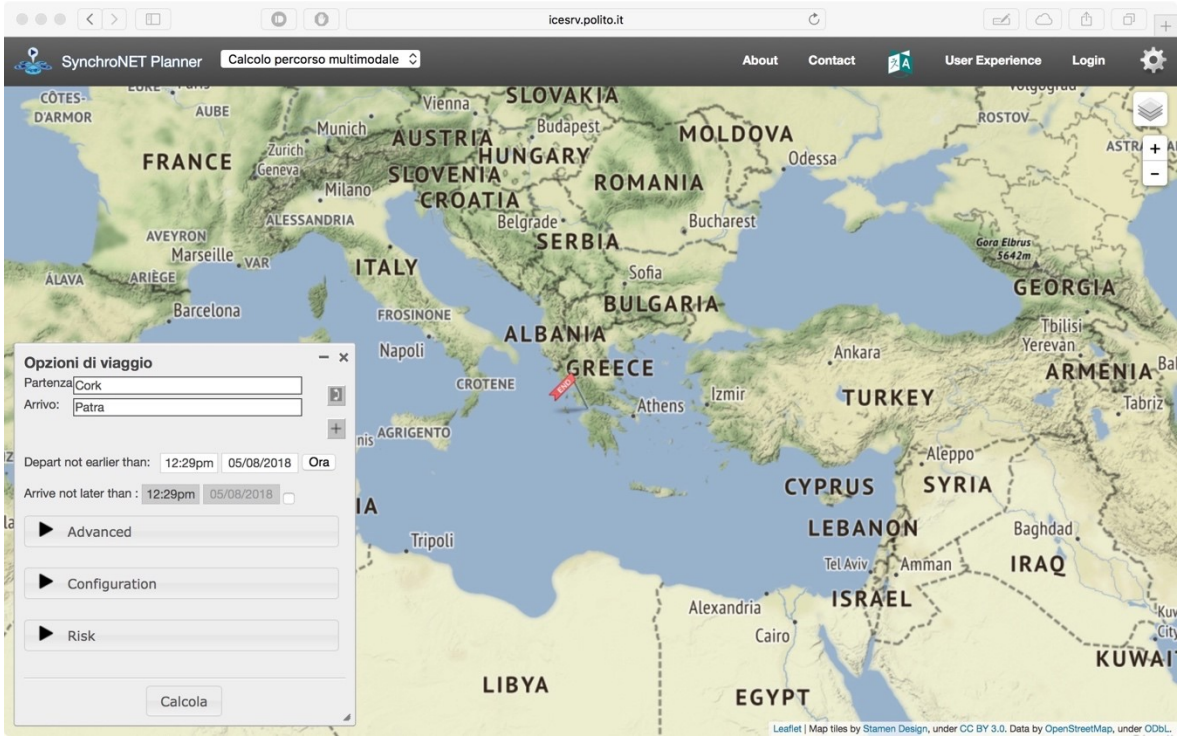


Figura 8 – Inserite Cork e Patrasso come partenza e destinazione

Nella figura 9 possiamo vedere il sommario della ricerca richiesta da Cork a Patrasso. Nella prima colonna "**Transport Modes**" possiamo notare quale tipo di trasporto verrà utilizzato in ciascuno degli itinerari; la seconda colonna "**Quality**" da un valore di qualità degli

itinerari trovati, con il primo itinerario a 100 e l'ultimo a 1 e il resto delle percentuali è in proporzione dal primo itinerario. La terza colonna (quella in blu) indica la **durata** dell'itinerario; la quarta le **emissioni** (in chilogrammi di CO2 emessa) e la quinta la **lunghezza** in chilometri. Dopo troviamo le colonne sul **costo totale**, quante **fermate** ha l'itinerario e la **durata solo di viaggio** escluse le soste.

Nel primo caso (che verrà analizzato nel dettaglio in seguito) possiamo constatare che il tragitto prevede l'utilizzo di truck, treni e navi; non è il percorso più breve (in lunghezza) ma tra i costi totali, e i tre parametri colorati è il migliore insieme al secondo itinerario.

Come si può constatare dall'immagine il software permette anche la visualizzazione grafica del percorso sulla mappa: i tratti in nave sono di colore blu, quelli in treno giallo e infine quelli in truck in verde.

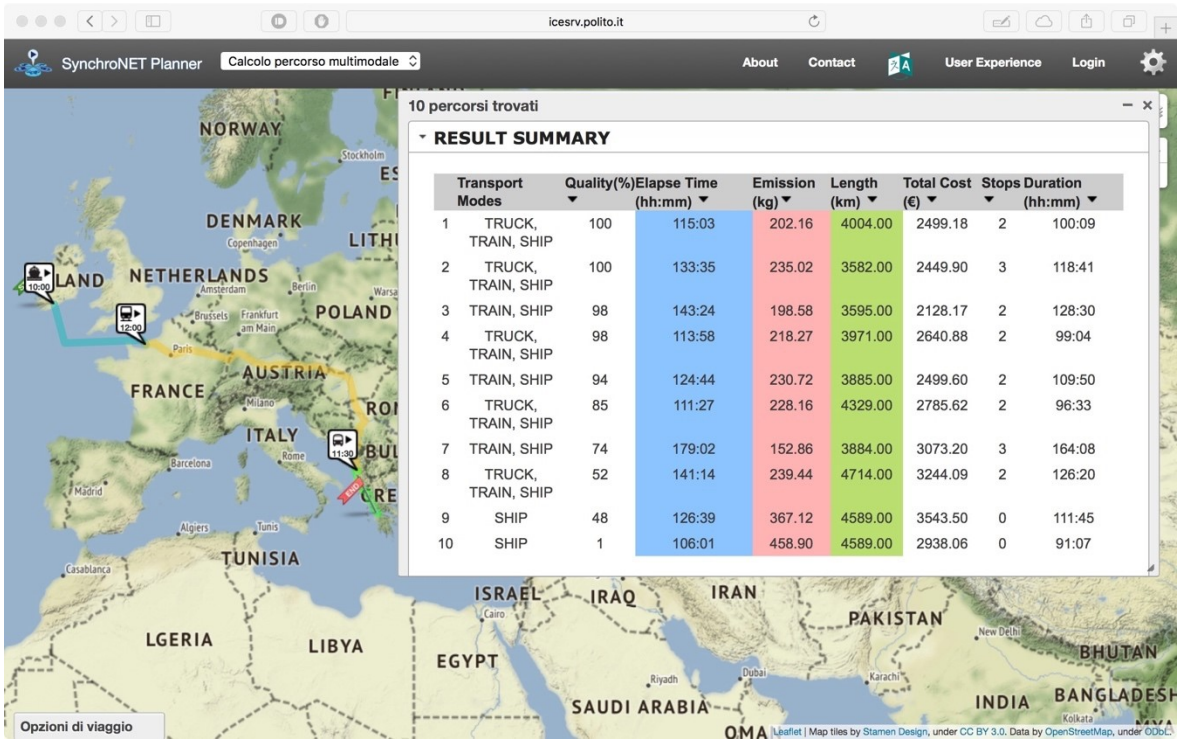
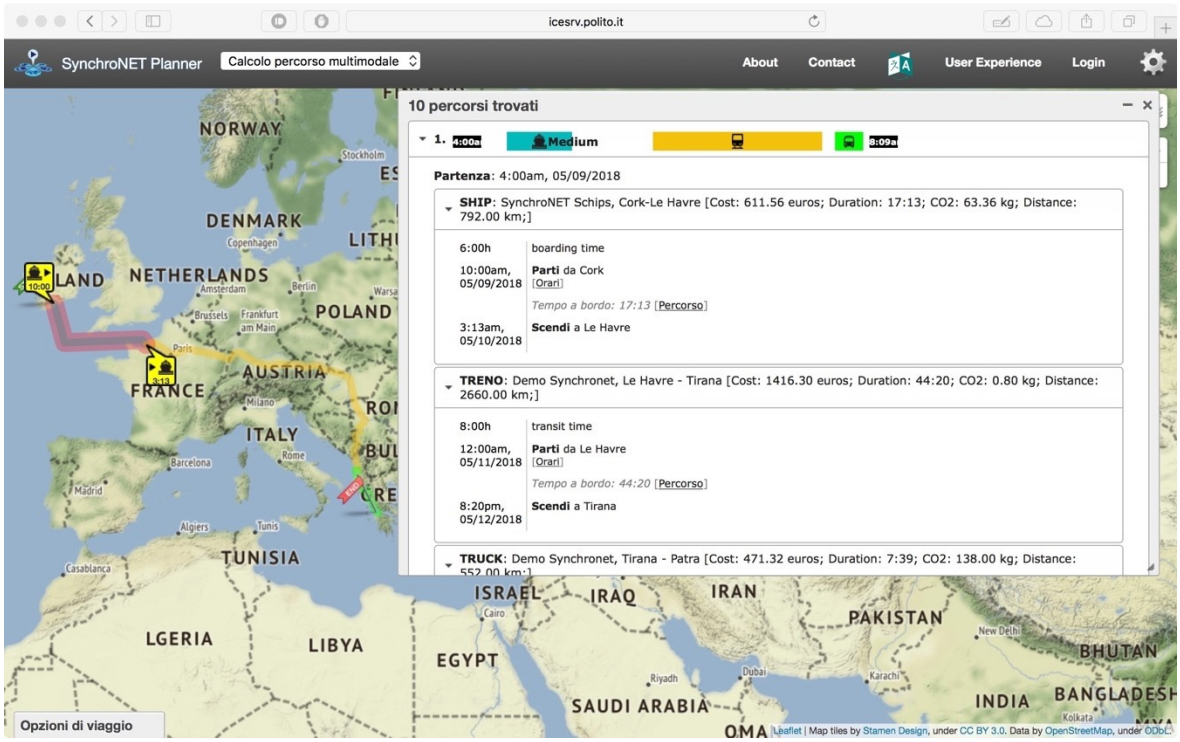


Figura 9 – Sommario risultati ricerca Cork Patrasso



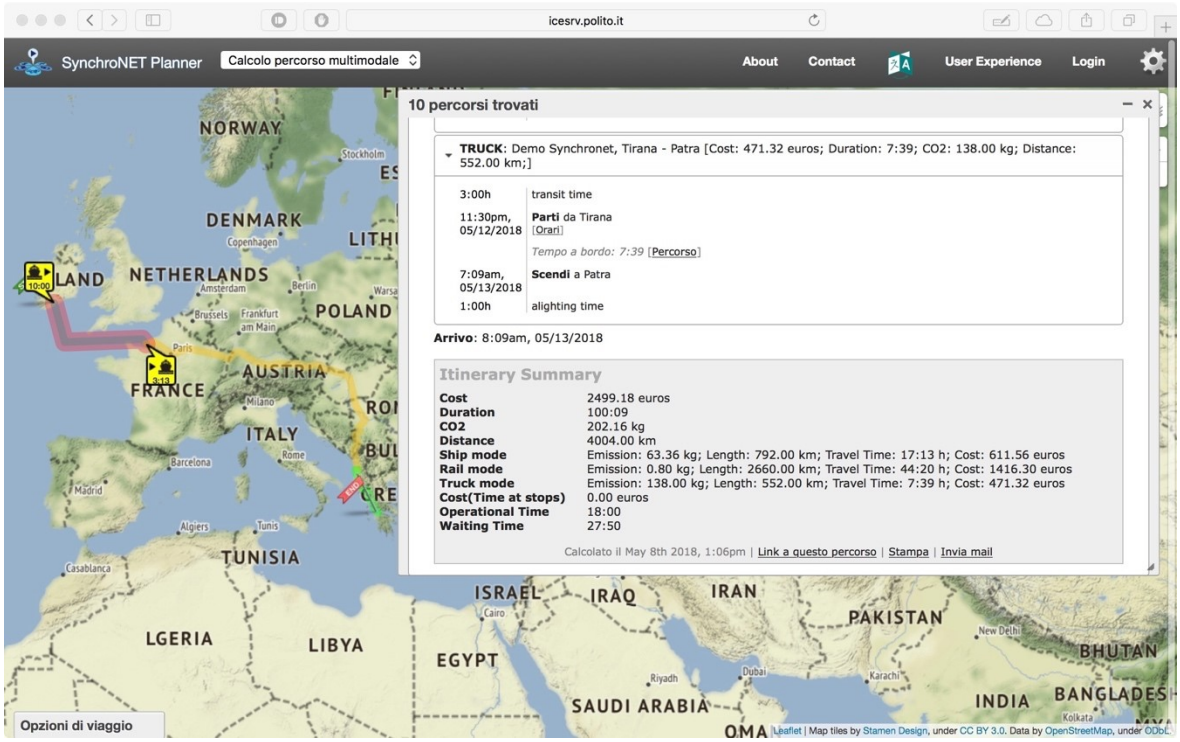


Figure 10a e 10b - Analisi del primo itinerario della tratta Cork Patrasso

Nelle figure 10a e 10b possiamo analizzare il **risultato** ottenuto dal primo itinerario: ogni itinerario è diviso in **tratte** che compongono l'intero itinerario. In questo caso l'itinerario è composto da **tre tratte**: Cork – Le Havre in **nave**; Le Havre – Tirana in **treno** e Tirana – Patrasso tramite **truck**. Entrando nel dettaglio della prima tratta Cork – Le Havre: l'output ci dice che la tratta è in nave; il **costo** della singola tratta è di 611,56 euro; della **durata** di 17 ore e 13 minuti; 63,36 kg di **CO2** emessi e della **distanza** di 792 km. In più ci viene fornito che la **partenza** da Cork è alle 12:00 con l'orario di **imbarco** alle 6:00 del 09/05/2018 con **l'arrivo** a Le Havre alle 3:13 del 10/05/2018.
La seconda tratta è Le Havre – Tirana: la tratta è in treno; il costo della singola tratta è di 1416,30 euro; della durata di 44 ore e 20 minuti; 0,80 kg di CO2 emessi e della distanza di 2660 km. In più ci viene fornito che la partenza da Le Havre è alle 12:00 del 11/05/2018 con l'arrivo a Tirana alle 20:20 del 12/05/2018.
La terza ed ultima tratta è Tirana – Patrasso: la tratta è in truck; il costo della singola tratta è di 471,32 euro; della durata di 7 ore e 39 minuti; 138 kg di CO2 emessi e della distanza di 552 km. In più ci viene fornito che la partenza da Tirana è alle 12:00 del 11/05/2018 con l'arrivo a Tirana alle 20:20 del 12/05/2018.
In fine dopo l'ultima tratta troviamo un **sommario** dell'itinerario con le seguenti informazioni:

- il costo totale dell'itinerario (in questo caso 2499,18 euro);
- la durata (100 ore e 9 minuti);
- la CO2 emessa (202,16 kg);
- la distanza totale (4004 km);
- le informazioni di tutte le tratte in nave presenti nell'itinerario (emissioni di sole navi, lunghezza, tempo di percorrenza e costo);
- le informazioni di tutte le tratte in treno presenti nell'itinerario;
- le informazioni di tutte le tratte in truck presenti nell'itinerario;
- costi nei momenti intermedi tra una tratta ed un'altra;
- tempi operativi;
- tempi totali di fermo.

1.5.1.2 Multi destination mode

La **multi destination mode** funziona come la single destination mode ma come il nome suggerisce con una **multipla** destinazione: dalla partenza si cerca un itinerario che abbia almeno una tratta in comune con tutti gli itinerari verso le altre destinazioni.

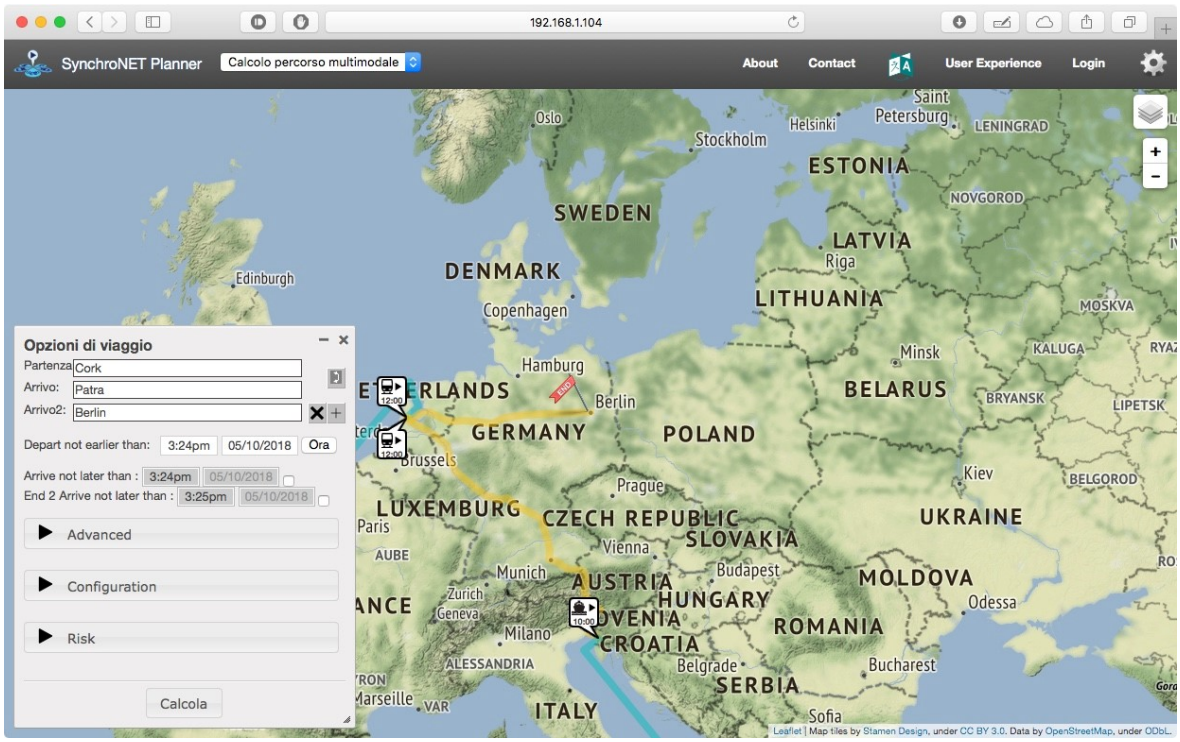
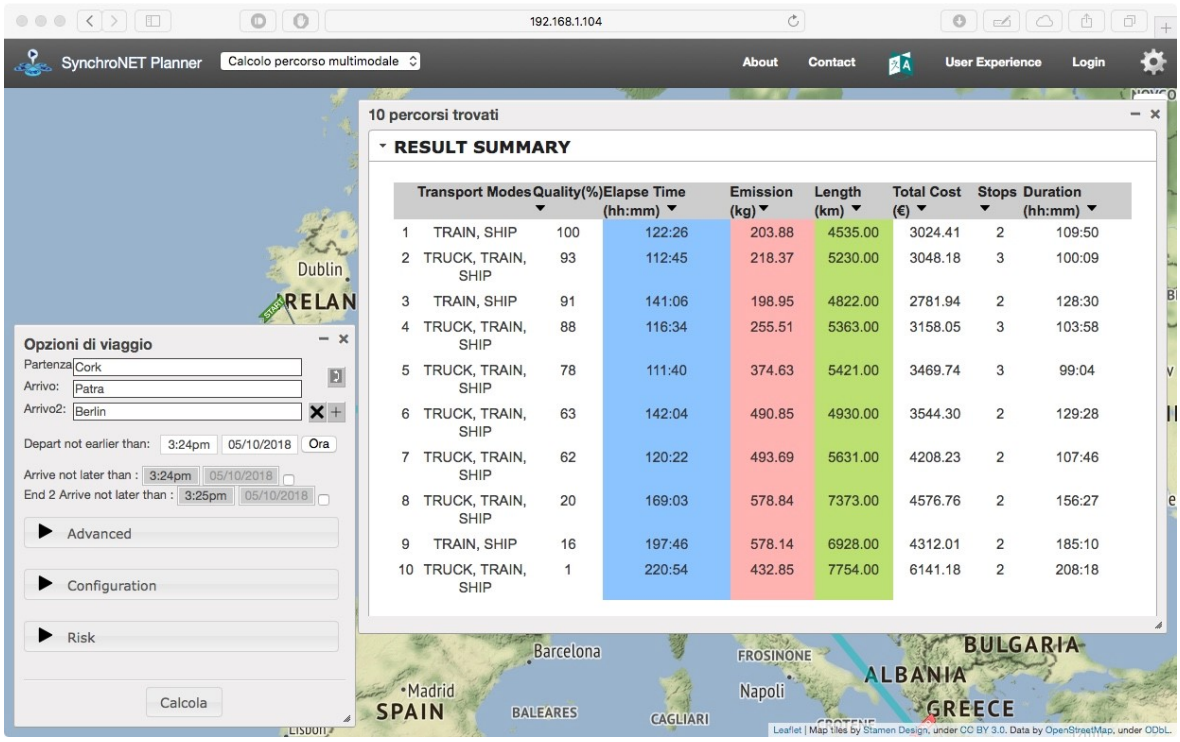


Figura 11 – Partenza Cork, destinazioni: Patrasso e Berlino

Nella figura 11 si può notare l'inserimento del campo "Arrivo2" tramite il pulsante "+" alla destra del campo Arrivo. In questo esempio abbiamo inserito **Cork** come partenza e **Patrasso** e **Berlino** (Germania) come città di arrivo. Come nella modalità single mode si può settare la data e l'ora ad entrambi gli arrivi.



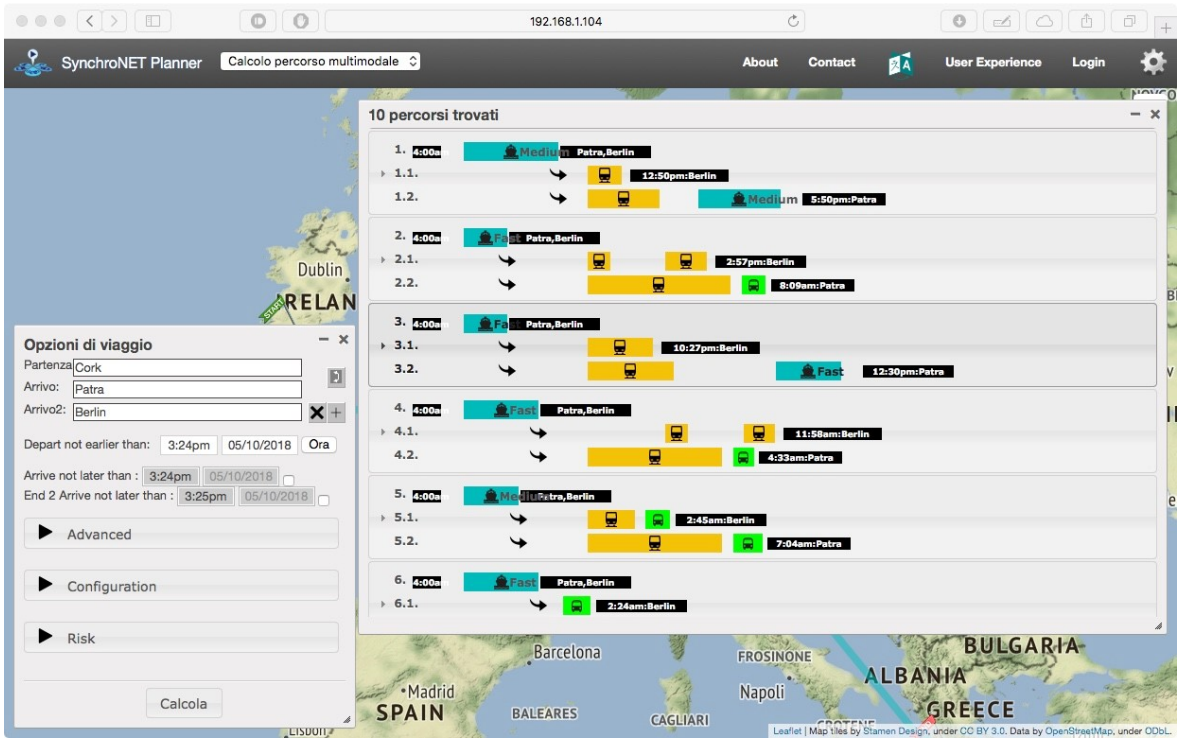


Figure 12 e 13 – Risultati Cork, Patrasso Berlino

Nelle figure 12 e 13 possiamo constatare i **percorsi** trovati dal simulatore: come si nota immediatamente l'output grafico è identico al single destination mode per il sommario dei risultati: invece per ogni singolo percorso possiamo notare che al livello grafico ci sono le etichette con la città di destinazione finale e le frecce per indicare il momento di divisione interna tra le diverse destinazioni finali.

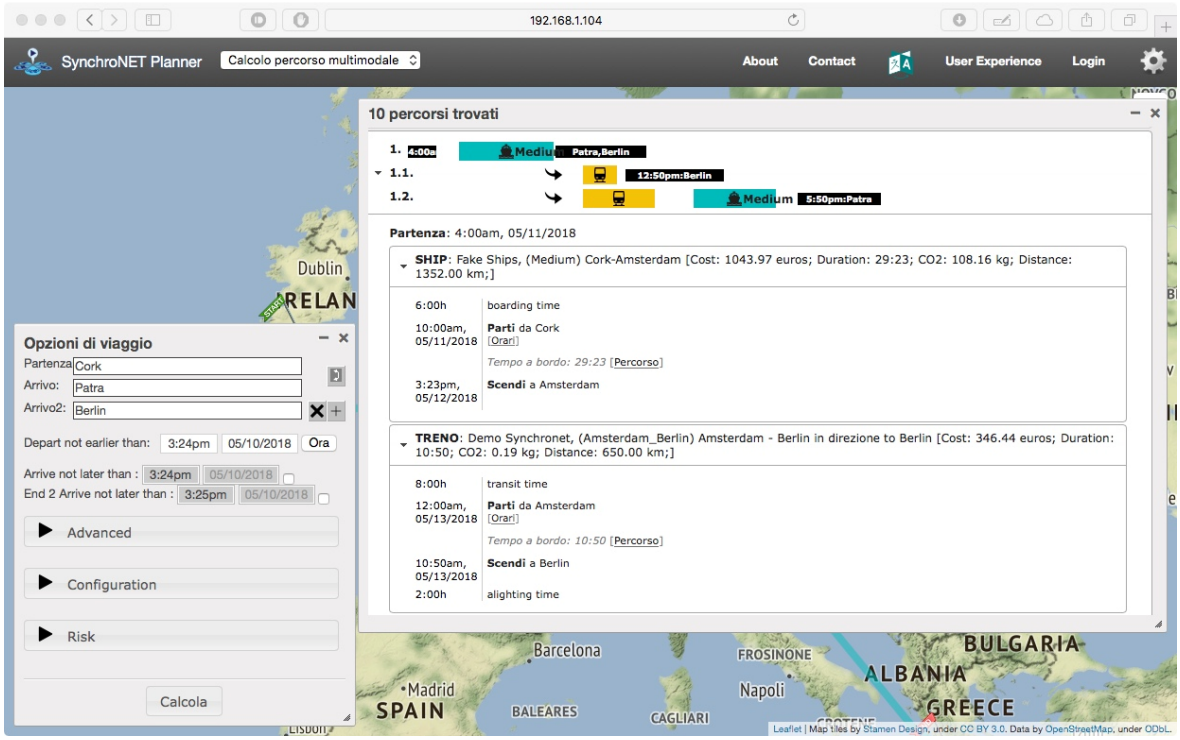
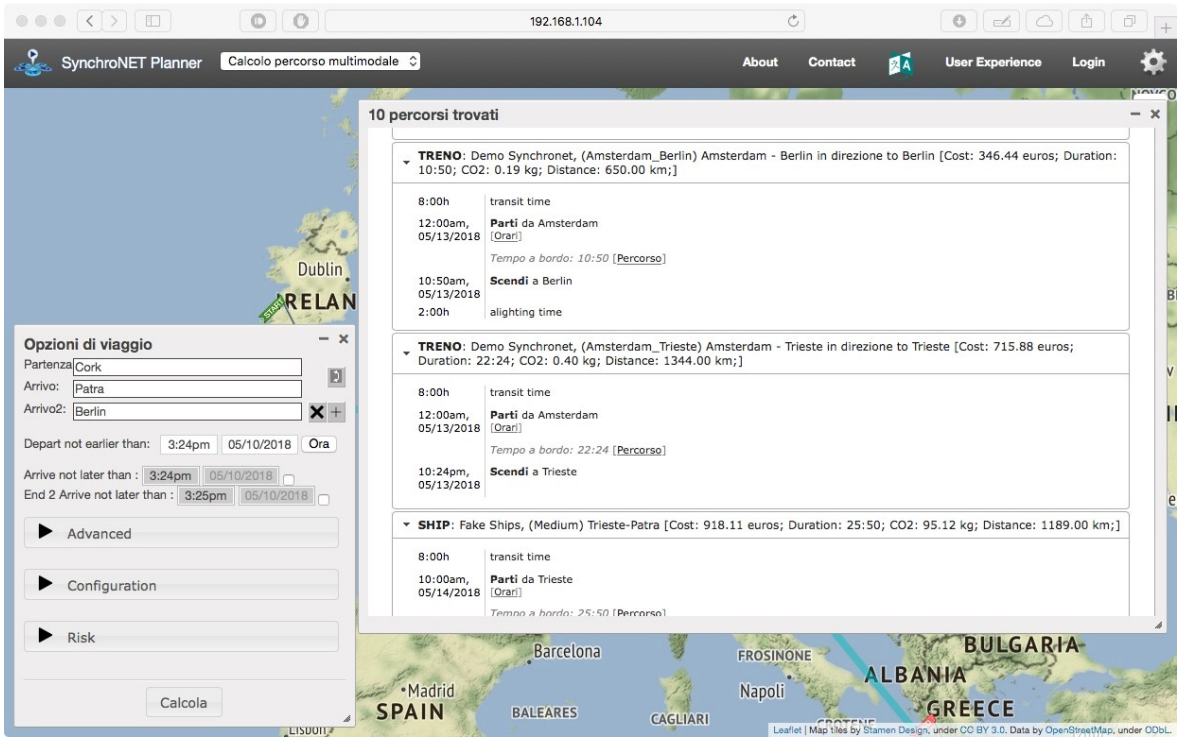


Figura 14a – Analisi risultato primo itinerario Cork Patrasso Berlino

Dalla figura 14 (a, b, c) possiamo analizzare il primo itinerario che il simulatore da nell'output: la **tratta in comune** tra le due destinazioni finali è solo **la prima**; codesta tratta parte da Cork e arriva ad Amsterdam; successivamente la seconda tratta in ordine è Amsterdam – Berlino in direzione Berlino; le altre due tratte sono in direzione Patrasso e sono Amsterdam – Trieste e Trieste – Patrasso.

La prima tratta è Cork – Amsterdam: la tratta è in nave; in direzione Berlino, Patrasso; il costo della singola tratta è di 1043,97 euro; della durata di 29 ore e 23 minuti; 108,16 kg di CO2 emessi e della distanza di 1352 km. In più ci viene fornito che la partenza da Cork è alle 10:00 del 11/05/2018 con l'arrivo ad Amsterdam alle 15:23 del 12/05/2018.

La seconda tratta è Amsterdam – Berlino: la tratta è in treno; in direzione Berlino; il costo della singola tratta è di 346,44 euro; della durata di 10 ore e 50 minuti; 0,19 kg di CO2 emessi



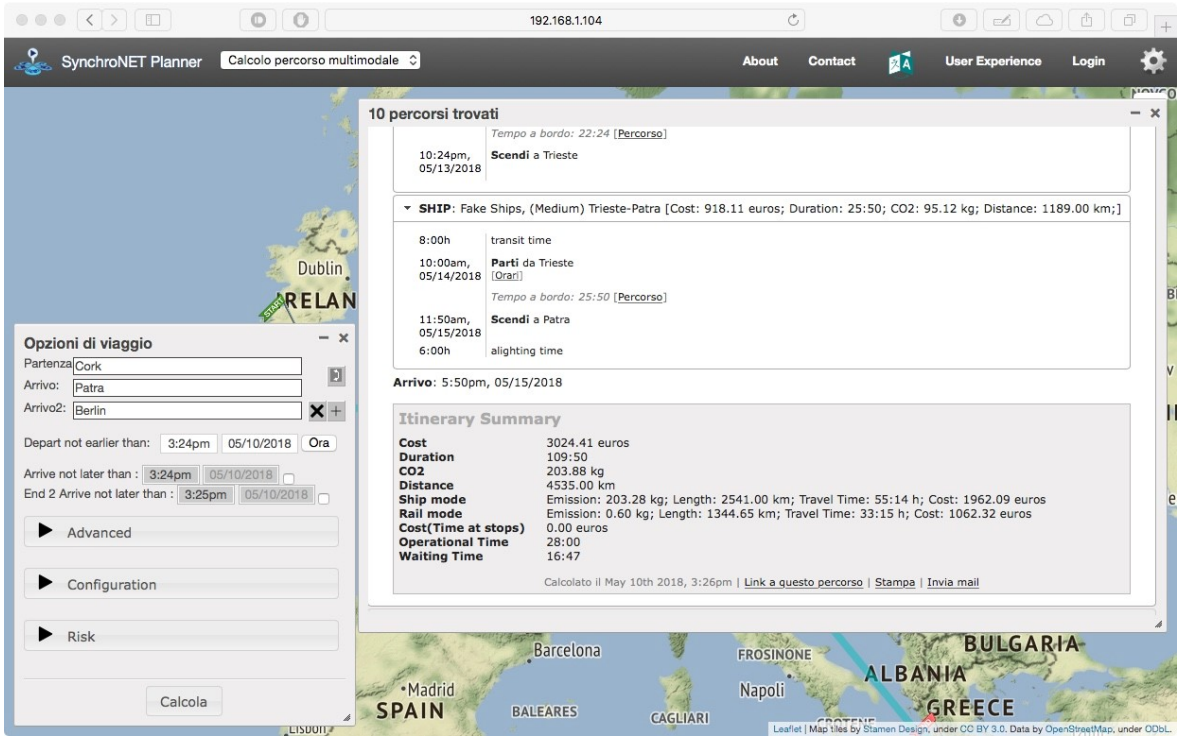


Figura 14b e 14c – Analisi risultato primo itinerario Cork Patrasso Berlino

e della distanza di 650 km. In più ci viene fornito che la partenza da Amsterdam è alle 00:00 del 13/05/2018 con l'arrivo a Berlino alle 10:50 del 13/05/2018. Questa tratta conclude la destinazione verso Berlino.

La terza tratta è Amsterdam – Trieste: la tratta è in treno; in direzione Patrasso; il costo della singola tratta è di 718,88 euro; della durata di 22 ore e 24 minuti; 0,40 kg di CO2 emessi e della distanza di 1344 km. In più ci viene fornito che la partenza da Amsterdam è alle 00:00 del 13/05/2018 con l'arrivo a Trieste alle 22:24 del 13/05/2018.

La quarta tratta è Trieste – Patrasso: la tratta è in nave; in direzione Patrasso; il costo della singola tratta è di 918,11 euro; della durata di 25 ore e 50 minuti; 95,12 kg di CO2 emessi e della distanza di 1189 km. In più ci viene fornito che la partenza da Trieste è alle 10:00 del 14/05/2018 con l'arrivo a Patrasso alle 11:50 del 15/05/2018: questa tratta conclude la destinazione verso Patrasso e l'intero itinerario.

Al termine delle tratte anche in questo caso troviamo il **sommario** dell'itinerario identico a quello visto per la modalità single destination mode.

2 Implementazione della “Capacity Mode”

Dopo aver introdotto il progetto SyncroNET e dopo aver illustrato il funzionamento dell’OTP, in questo capitolo verrà introdotta la “**Capacity Mode**” con le sue specifiche e come ho voluto implementarla.

2.1 “Capacity Mode”: che cos’è?

Come visto nel capitolo precedente OTP data una partenza e uno o più arrivi da come output un insieme di itinerari che soddisfano la richiesta iniziale. A questo punto si entra nel cuore di questa tesi: è possibile mandare un numero di **container**, dato come input, dal punto di partenza ai vari punti di arrivo? La risposta di questa domanda è la “Capacity Mode”.
Inoltre, l’obbiettivo della nuova implementazione è nel rendere il simulatore più **reale** possibile: quindi oltre a gestire il numero di container bisogna anche gestire la transazione da un veicolo ad un altro, i tempi di attesa, le emissioni, i rischi di certe tratte che possono venire meno come compromesso alla suddivisione dei container da spedire.
Tuttavia, non è così semplice introdurre una **modifica** così strutturale poiché bisogna tenere conto di molti fattori: i veicoli sono a capacità limitata, e quindi bisogna gestire l’eventuale carico che alcune tratte non possono portare o direttamente scartare alcuni itinerari; l’efficienza del simulatore con incluso il processo di questa “mode”: bisogna valutare l’impatto in termini di **prestazione** poiché ogni tratta che non soddisfa i requisiti andrebbe sostituita o parallelizzata con altre tratte.

2.2 Specifiche della “Capacity Mode”

Nello specifico la “Capacity Mode” funziona così: l’implementazione stessa si colloca al **termine** del simulatore; cioè esso inizia il suo processo senza considerare il numero di container da inviare, e crea una lista di oggetti Itinerary che senza la “Capacity Mode” poi vengono visualizzati direttamente nell’output. Prima di mandare questa lista come output finale entra in gioco la classe “CapacityCalculator” che prende la suddetta **lista** e la **modifica** per effettuare il **calcolo dei container** da mandare dal punto di partenza ai punti di arrivo.

Come spiegato nel capitolo 1.3.1 l’OTP si basa su dei **dati aperti** e in questo caso nei GTFS oltre alle informazioni sugli orari e i costi, troviamo anche le informazioni sulle **capacità** delle singole rotte.

Dato che in questo momento i GTFS non contengono totalmente tutti i dati di capacità di ogni singola rotta, l’OTP da dei **valori di default** per tipologia di trasporto: alle rotte in nave viene assegnato 1000 container di capacità, a quelle in treno 200 e i truck vengono considerati con capacità illimitata poiché ogni truck porta un solo container ma possono viaggiare in flotte infinite (a livello teorico). I calcoli dei **costi** e delle emissioni vengono **proporzionati** in base ai container totali che le navi e i treni possono trasportare, invece per i truck vengono **moltiplicati** direttamente per il numero di container.

Ogni singolo itinerario è un oggetto di tipo **Itinerary** ed è formato da più rotte che sono oggetti di tipo **Leg**: quindi un oggetto Itinerary ha una lista di Leg che forma il suddetto itinerario. Come visto al termine del capitolo precedente i costi di ogni singolo itinerario sono la somma dei costi delle tratte che lo formano.

Al principio la “Capacity Mode” doveva funzionare in questo modo: in prima battuta la classe “Capacity Calculator” controlla che i singoli itinerari trovati dall’OTP soddisfino i criteri di ricerca dei container; in secondo luogo se almeno una tratta soddisfa l’input allora si trovano soluzioni alternative per il trasporto dei container mancanti e come ultima opzione unire due itinerari che non soddisfano le condizioni ma che sommati insieme riescono a trasportare tutti i container.

Quest’ultima soluzione in fase finale è stata eliminata poiché in caso di destinazione multipla non era implementabile, e quindi dato che la seconda opzione riesce, in caso di single destination mode, a soddisfare con un numero **significativo** di itinerari si è ritenuto di toglierla per aumentare le prestazioni di tempo di calcolo.

Se di un itinerario neanche una singola tratta non riesce a soddisfare la richiesta iniziale, questo itinerario viene scartato a priori poiché questo comporterebbe la ricerca di soluzioni alternative ad ogni singola tratta che comporterebbe sicuramente ad un esito qualitativo di scarso livello.

In principio la “Capacity Mode” era pensata solo per la modalità single destination, ma successivamente è stata **adottata e modificata** per funzionare con entrambe le modalità: per fare in modo di avere un unico algoritmo, in modo di funzionare sia con la single destination che con la multiple destination, si è deciso di **spezzare** l'itinerario completo (della multi destination) in sotto itinerari per tratte comuni e singole, facendo in modo di avere un pseudo itinerario che può essere trattato in maniera identica di un itinerario della single destination.

Ciò è stato reso possibile dall'attributo che ogni oggetto di tipo Leg possiede: un'etichetta dove viene indicato di quali destinazioni una determinata Leg fa parte. Infatti, vengono considerati un unico pseudo itinerario tutte le tratte che hanno la stessa etichetta.
A livello di codice come poi verrà analizzato in seguito bisogna prima di tutto instanziare l'oggetto di tipo *CapacityCalculator* passandogli come parametro un oggetto di tipo *SpecialRouteFinder*; e successivamente invocare il metodo *startCapacityMode* con tutti i vari parametri (vedi al paragrafo successivo). A partire da questo *start* poi verranno chiamati i vari metodi (figura 15).

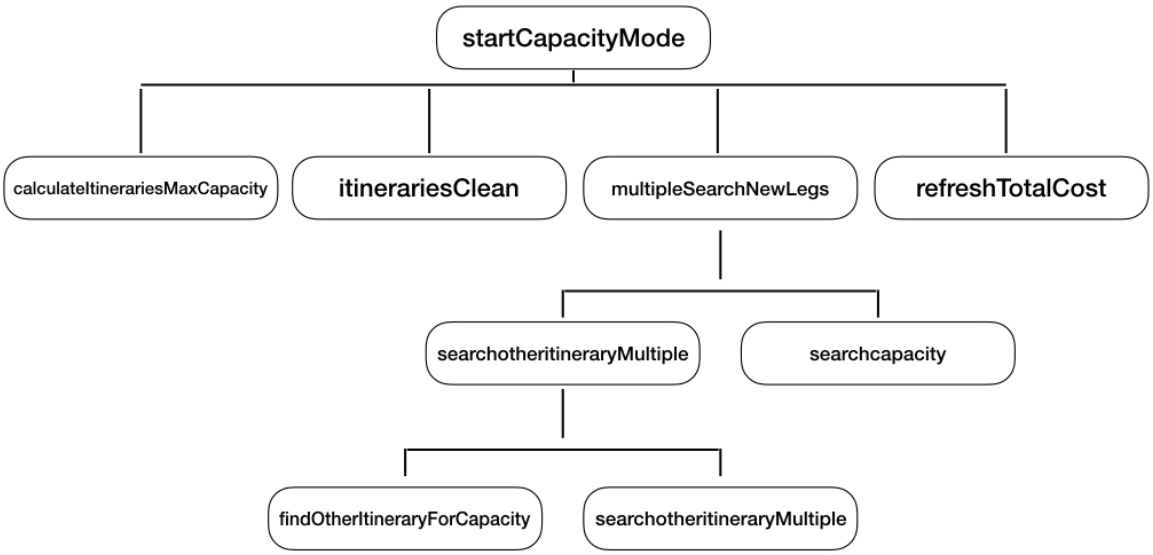


Figura 15 – Grafo classe *CapacityCalculator*

2.3 Framework Operativo

Per la realizzazione della "Capacity Mode" è stata creata la classe *CapacityCalculator* nel package OTP scritto in linguaggio **Java**. È stato utilizzato l'ambiente di sviluppo Eclipse[8] per lo sviluppo importando il progetto esistente di tipo Maven, MySQL[9] per l'utilizzo del database creando e aggiungendo una tabella con il nome "synchronet".
Ovviamente utilizzando Java come linguaggio di programmazione bisogna aver installato il JDK (Java Development Kit) per l'esecuzione del progetto tramite la Java Virtual Machine.

[8] Eclipse sito web: https://www.eclipse.org (accesso il 29/06/18)
[9] MySQL sito web: https://www.mysql.com (accesso il 29/06/18)

Inoltre, è stato utilizzato il linguaggio di programmazione R[10] per l’analisi statistica dei dati unito a Java.
Per lo sviluppo di codice con il resto del team è stato utilizzata la piattaforma GitLab[11] per la condivisione e la collaborazione del codice.

2.4 Analisi del codice implementato

In questa sezione vedremo in dettaglio l’implementazione della “Capacity Mode” soprattutto la classe *CapacityCalculator* che sta alla **base** della modalità.
Come spiegato nella sezione precedente la classe *CapacityCalculator* prende la lista di itinerari, la modifica e la ritorna all’OTP; entra in gioco nella classe *PlannerResource* dell’OTP. Questa è l’implementazione all’interno della suddetta classe:

```
CapacityCalculator capacityCalculator = new CapacityCalculator(srf);
int destinationsNumber = 1;
Date[] arrivaltime;
int[] multipleCapacity;
if(otherDestinations != null) {
//Multi-Destination
destinationsNumber += otherDestinations.length;
multipleCapacity = new int[destinationsNumber];
arrivaltime = new Date[destinationsNumber];

multipleCapacity[0] = request.capacityRequest;
String multiCapacity = response.requestParameters.get("otherCapacityRequest");
                    String[] otherCapacity = multiCapacity == null ? null :
multiCapacity.split("---");

for(int i = 1; i < destinationsNumber; i++) {
//for each destination save the arrival time and the capacity
      if(hasArrivalTimes[i-1])
            arrivaltime[i] = DateUtils.toDate(otherDates[i-1], otherTimes[i-1],
TimeZone.getDefault());

            multipleCapacity[i] = Integer.parseInt(otherCapacity[i-1]);
            if(multipleCapacity[i] <= 0)
```

[10] R Project sito web: https://www.r-project.org (accesso il 29/06/18)
[11] GitLab sito web: https://about.gitlab.com (accesso il 29/06/18)

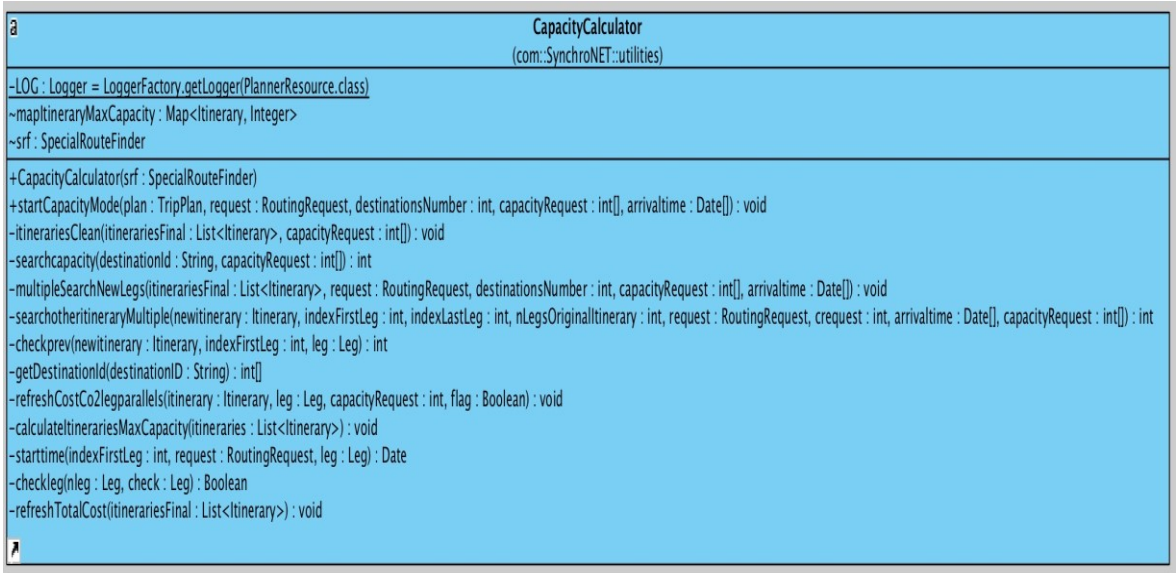
```
                                multipleCapacity[i] = 1;
        }

} else {
//Single Destination
multipleCapacity = new int[1];
arrivaltime = new Date[1];
}


//save arrival time and capacity of the first destination if it has one
multipleCapacity[0] = request.capacityRequest;

if(multipleCapacity[0] <= 0) {
        multipleCapacity[0] = 1;
}

if(request.timeArriveVisible)
        arrivaltime[0] = request.getDateTimeArrive();


capacityCalculator.startCapacityMode(plan, request, destinationsNumber,
multipleCapacity, arrivaltime);
```

I **vari input** dati dall'utente con il numero di container l'OTP lo inserisce in una **unica variabile**: per ogni destinazione si inserisce il numero di container e nelle tratte comuni il numero di container trasportato risulta la **somma** dei container che devono giungere le varie destinazioni. Quindi viene creato un **array** di interi con i vari valori dei container che devono essere spediti per ogni destinazione; in aggiunta viene creato un array di *Date* con all'interno i vari orari di arrivo delle varie destinazioni (se settate).

2.4.1 CapacityCalculator

In questa sezione analizzeremo la classe *CapacityCalculator* con i suoi vari metodi.

2.4.1.1 Class Diagram



2.4.1.2 Costruttore

```
public class CapacityCalculator {

	private static final Logger LOG =
LoggerFactory.getLogger(PlannerResource.class);

	Map<Itinerary, Integer> mapItineraryMaxCapacity;
	SpecialRouteFinder srf;

	public CapacityCalculator(SpecialRouteFinder srf) {

		mapItineraryMaxCapacity = new ConcurrentHashMap<>();
		this.srf = srf;

	}
```

Come si può notare il costruttore della classe *CapacityCalculator* riceve un solo parametro che è di tipo *SpecialRouteFinder*, che viene utilizzato come variabile globale in questa classe e in più viene inizializzato un oggetto di tipo *ConcurrentHashMap* che è una mappa (utilizzata come variabile globale) di tipo *Itinerary-Integer* che verrà riempita più avanti.

2.4.1.3 startCapacityMode

```
public void startCapacityMode(TripPlan plan, RoutingRequest request, int
destinationsNumber, int[] capacityRequest, Date[] arrivaltime) {

            if (plan == null || plan.itinerary.isEmpty()) return;

            List<Itinerary> itinerariesFinal = new ArrayList<>();
            calculateItinerariesMaxCapacity(plan.itinerary); // create a Map
with Itinerary and his max capacity.

            LOG.info("START CAPACITY");
            LOG.info("Destinations Number: " +
Integer.toString(destinationsNumber));
            itinerariesClean(itinerariesFinal, capacityRequest);
            multipleSearchNewLegs(itinerariesFinal, request,
destinationsNumber, capacityRequest, arrivaltime);
            // if(multiple==1)
            // merge2itinerary(request.capacityRequest, itinerariesFinal);
            refreshTotalCost(itinerariesFinal);
            plan.itinerary =
QualityCalculator.calculateItinerariesQualityListSort(itinerariesFinal,
request);

      }
```

StartCapacityMode è il metodo di **partenza** della "Capacity Mode" dove gli viene passato come parametro l'elemento *TripPlan* dove l'OTP ha la lista degli itinerari, *RoutingRequest* (verrà spiegato in seguito il suo utilizzo), *destinationsNumber* che contiene il numero di destinazioni con un vettore di interi con il numero di container da spedire per ciascuna destinazione e infine un array di tipo *Data* con le date di arrivo per ogni singola destinazione. Questo metodo in prima battuta **controlla** che la lista di itinerari non sia vuota, successivamente lancia il metodo *calculateItinerariesMaxCapacity* che si occupa si calcolare per ogni itinerario la capacità minima trasportata, dopo con *itinerariesFinal* controlla se gli itinerari **soddisfano** già le richieste. Successivamente viene lanciato il metodo *multipleSearchNewLegs* che verrà spiegato più avanti. Alla fine con il metodo *refreshTotalCost* vengono aggiornati i **costi** di tutti gli itinerari e con `QualityCalculator.calculateItinerariesQualityListSort` gli itinerari trovati dalla

“Capacity Mode” vengono passati alla classe *QualityCalculator* dove verrà calcolata la **qualità** e riassegnata la lista di itinerari all’OTP tramite l’elemento *plan*.

2.4.1.4 calculateItinerariesMaxCapacity

```
private void calculateItinerariesMaxCapacity(List<Itinerary> itineraries) {
		LOG.info("Finding max capacity for each itinerary");
		if (itineraries.isEmpty()) {
			LOG.error("listItineraries is empty");
			return;
		}

		for (Itinerary itinerary : itineraries) {
			//calculate max capacity for each itinerary
			List<Leg> listLeg = itinerary.legs;
			if (listLeg.isEmpty()) {
				LOG.warn("listLeg is empty");
			}

			int maxCapacity = Integer.MAX_VALUE;
			for (Leg leg : listLeg) {
				//if the mode is not truck and the maxCapacity of the
itinerary is higher then the leg capacity
				//update the new max capacity
				if (!leg.isTRUCKLeg() && maxCapacity > leg.capacity)
						maxCapacity = leg.capacity;
			}

			mapItineraryMaxCapacity.put(itinerary, maxCapacity);

		}

	}
```

Come spiegato precedentemente questo metodo **popola** la mappa *mapItineraryMaxCapacity* con ogni itinerario con il suo valore minimo di container che può trasportare dato dalla tratta con la capacità minore, considerando sempre i truck a capacità illimitata.

2.4.1.5 itinerariesClean

```
	//Trovo itinerario unico
	private void itinerariesClean(List<Itinerary> itinerariesFinal, int[]
capacityRequest) {
		for (Map.Entry<Itinerary, Integer> itinerary :
mapItineraryMaxCapacity.entrySet()) {
			// Ho cercato l'itinerario che ha già la capacità
disponibile.
			Boolean ok = true;
			for (Leg leg : itinerary.getKey().legs) {
				int capacityforleg = searchcapacity(leg.destinationId,
capacityRequest);
				if (!leg.isTRUCKLeg())
					if (leg.capacity < capacityforleg) {
						ok = false;
						break;
					}
			}
			if (ok) {
				for (Leg leg : itinerary.getKey().legs) {
					leg.assignedcontainers =
searchcapacity(leg.destinationId, capacityRequest);
					//refreshCostCo2leg(itinerary.getKey(), leg,
true, leg.capacity);
				}
				itinerariesFinal.add(itinerary.getKey());
				mapItineraryMaxCapacity.remove(itinerary.getKey());
			}
		}

	}
```

Come spiegato in precedenza questo metodo si occupa di verificare se gli itinerari forniti dall'OTP **soddisfano** la richiesta di container richiesta: se l'itinerario è già pronto verrà inserito nella lista *itinerariesFinal* e rimossa dalla mappa *mapItineraryMaxCapacity* che è stata popolata in *calculateItinerariesMaxCapacity*.

2.4.1.6 *searchcapacity*

```
private int searchcapacity(String destinationId, int[] capacityRequest) {
		StringTokenizer st = new StringTokenizer(destinationId, "-");
		int capacity = 0;
		while (st.hasMoreElements()) {
			int number = Integer.parseInt(st.nextElement().toString());
			number--;
			capacity += capacityRequest[number];
		}
		return capacity;
}
```

Nel caso di multiple destination mode, per ogni tratta (leg) viene settata un’unica variabile di tipo *String* con tutte le destinazioni (interi) separate da un carattere ‘-‘. L’informazione dei container da spedire è nel vettore *capacityRequest*, e al termine viene restituito un intero con la **somma** di tuti i container che devono passare per quella tratta (cioè la somma dei container diretti a tutte le destinazioni che quella tratta porta).

2.4.1.7 *multipleSearchNewLegs*

```
private void multipleSearchNewLegs(List<Itinerary> itinerariesFinal,
RoutingRequest request, int destinationsNumber, int[] capacityRequest, Date[]
arrivaltime) {
	for (Map.Entry<Itinerary, Integer> itinerary :
mapItineraryMaxCapacity.entrySet()) {
	LOG.info("New itinerary");
	Boolean maxcapacity = false;
	Map<Integer, Boolean> legscheck = new ConcurrentHashMap<>();
	int count = 0;
	for (Leg leg : itinerary.getKey().legs) {
		if (leg.isTRUCKLeg()) {
			maxcapacity = true;
		} else {
			if (leg.capacity >= searchcapacity(leg.destinationId,
capacityRequest))
				maxcapacity = true;
		}
		legscheck.put(count, false);
		count++;
```

```
			}
		if (maxcapacity) {
			LOG.info("   Now try to search other leg to add");
			Itinerary newitinerary = new Itinerary();
			newitinerary.cloneItinerary(itinerary.getKey(), true);
			int sizeNewiniterary = newitinerary.legs.size();
			Boolean check = true;
			for (int indexLegs = 0; indexLegs < sizeNewiniterary; indexLegs++)
{
				if (!legscheck.get(indexLegs)) {
					Leg leg = newitinerary.legs.get(indexLegs);
					int crequest = searchcapacity(leg.destinationId,
capacityRequest);
					LOG.info("Leg: " + leg.from.name + "-" + leg.to.name +
" " + leg.mode);
					LOG.info("Capacity request: " +
Integer.toString(crequest));
					LOG.info("Capacity leg: " +
Integer.toString(leg.capacity));
					if (!leg.isTRUCKLeg()) {
						if (leg.capacity < crequest) {
							int ret =
searchotheritinerarayMultiple(newitinerary, indexLegs, indexLegs,

		sizeNewiniterary, request, crequest, arrivaltime, capacityRequest);
							if (ret > 0) {

		legscheck.replace(indexLegs, true);
							if (ret > indexLegs) {
								int k = indexLegs;
								while (k <= ret) {
									if
(newitinerary.legs.get(k).destinationId.equals(leg.destinationId))
								legscheck.replace(k, true);
								k++;
										}
										}
								} else {

		LOG.info("searchotheritinerary non found");
								check = false;
```

```
                                        break;
                                              }
                                        } else {
                                        leg.assignedcontainers = crequest;
                                        }
                                  } else {
                                        leg.assignedcontainers = crequest;
                        //refreshCostCo2leg(newitinerary, leg, false, leg.capacity);
                                  }
                            }
                      }
                      if (check) {
                            itinerariesFinal.add(newitinerary);
                      }
                } else {
                      LOG.info("No, this itinerary is not good, all legs
have lower capacity than the requested one.");
                }

          }
    }
```

Il metodo *multipleSearchNewLegs* è il metodo che si occupa di **trovare** tratte alternative che soddisfino la richiesta iniziale viaggiando in parallelo alle tratte che non hanno abbastanza capacità per portare tutti i container richiesti. Gli itinerari che non vanno bene sono all’interno della mappa *mapItineraryMaxCapacity* e, selezionato un itinerario, prima di tutto bisogna controllare che almeno una tratta dello stesso riesca a portare tutti i container richiesti. Se l’itinerario **rispetta** le specifiche viene creato un nuovo oggetto itinerario e viene clonato; successivamente viene lanciato il metodo *searchotheritineraryMultiple* per **cercare** una tratta parallela a quella mancante. Se l’itinerario al termine è completo e soddisfa la capacità richiesta viene **aggiunto** alla lista *itinerariesFinal*.

2.4.1.8 searchotheritineraryMultiple

```
      private int searchotheritineraryMultiple(Itinerary newitinerary, int
indexFirstLeg, int indexLastLeg,
                  int nLegsOriginalItinerary, RoutingRequest request, int
crequest, Date[] arrivaltime, int[] capacityRequest) {

            int ultimo = 0;
```

```
            int penultimo = 0, prev = -1;
            List<Integer> lista = new ArrayList<>();
            for (int d = indexFirstLeg; d <= nLegsOriginalItinerary - 1; d++) {
                  if
(newitinerary.legs.get(d).destinationId.equals(newitinerary.legs.get(indexFirst
Leg).destinationId)) {
                        ultimo = d;
                        lista.add(d);
                  }

            }
            for (int next : lista) {
                  if (next < indexFirstLeg) {
                        prev = next;

                  }
            }
            if (lista.size() > 1)
                  penultimo = lista.get(lista.size() - 2);
            if (indexLastLeg < ultimo) {
                  for (int k = indexLastLeg; k < nLegsOriginalItinerary - 1;
k++) {
                        if (newitinerary.legs.get(k + 1).destinationId

      .equals(newitinerary.legs.get(indexFirstLeg).destinationId)) {
                              if (newitinerary.legs.get(k + 1).capacity <
crequest
                                          && !newitinerary.legs.get(k +
1).isTRUCKLeg()) {
                                    indexLastLeg = k + 1;
                              } else
                                    break;
                        }
                  }
            }
            int mincap = crequest;
            for (int m = indexFirstLeg; m <= indexLastLeg; m++) {
                  if
(newitinerary.legs.get(m).destinationId.equals(newitinerary.legs.get(indexFirst
Leg).destinationId)) {
                        if (newitinerary.legs.get(m).capacity < crequest &&
```

```
!newitinerary.legs.get(m).isTRUCKLeg())
                                    mincap = newitinerary.legs.get(m).capacity;
                        }
                  }
                  LOG.info("mincap " + Integer.toString(mincap));
                  for (int m = indexFirstLeg; m <= indexLastLeg; m++) {
                        if
(newitinerary.legs.get(m).destinationId.equals(newitinerary.legs.get(indexFirst
Leg).destinationId)) {
                              newitinerary.legs.get(m).assignedcontainers = mincap;
                              //refreshCostCo2leg(newitinerary,
newitinerary.legs.get(m), false, newitinerary.legs.get(m).capacity);
                              LOG.info("leg " + newitinerary.legs.get(m).from.name +
" "
                                          +
Integer.toString(newitinerary.legs.get(m).assignedcontainers));
                        }
                  }
                  Leg leg = newitinerary.legs.get(indexFirstLeg);
                  int[] destinationLeg = getDestinationId(leg.destinationId);
                  if (prev < 0 && indexFirstLeg > 0) {
                        prev = checkprev(newitinerary, indexFirstLeg, leg);
                  }
                  Leg arrive = newitinerary.legs.get(indexLastLeg);
                  LOG.info("cerco un nuovo itinerario in sostituzione");
                  LOG.info("Partenza " + leg.from.name);
                  LOG.info("Arrivo " + arrive.to.name);
                  String departure = leg.from.name + "::" + leg.from.lat + "," +
leg.from.lon;
                  String arrival = arrive.to.name + "::" + arrive.to.lat + "," +
arrive.to.lon;
                  Date arrivetime = null;
                  Date starttime = null;
                  Boolean movetruck = false;
                  int nt = 0, nntt = 0;
                  int indice = 0;
                  for (int next : lista) {
                        if (next > indexLastLeg) {
                              nt = next;
                              break;
                        }
```

```
                indice++;
            }
            if (lista.size() > indice + 1)
                nntt = lista.get(indice + 1);

            if (indexLastLeg < ultimo) {

                Leg nextleg = newitinerary.legs.get(nt);
                if (nextleg.isTRUCKLeg()) {
                    movetruck = true;
                    if (indexLastLeg < penultimo) {
                        Long timeadd =
newitinerary.legs.get(nntt).startTime.getTime().getTime()
                                -
nextleg.endTime.getTime().getTime();
                        arrivetime = nextleg.startTime.getTime();

      arrivetime.setTime(nextleg.startTime.getTime().getTime() + timeadd);
                    } else {
                        // movetruck = false;
                        if (destinationLeg.length == 1) {
                            if (arrivaltime[destinationLeg[0] - 1] !=
null)
                                arrivetime =
arrivaltime[destinationLeg[0] - 1];
                            } else
                                arrivetime = nextleg.startTime.getTime();
                        }
                } else {
                    arrivetime = nextleg.startTime.getTime();
                }
            } else {
                if (indexFirstLeg < newitinerary.legs.size() - 1) {
                    Boolean checkfinaldestination = true;
                    if (arrive.finalDestination > 0) {
                        String finaldest = ""+arrive.finalDestination;

      if(leg.capacity<searchcapacity(finaldest,capacityRequest)) {
                            arrivetime = arrive.endTime.getTime();
                            checkfinaldestination=false;
                        }
```

```
                        }
                        if (checkfinaldestination)
                              for (int search = indexLastLeg + 1; search <
newitinerary.legs.size() - 1; search++)
                                    if
(newitinerary.legs.get(search).previousDestinationId.equals(arrive.destinationI
d)) {
                                          arrivetime =
newitinerary.legs.get(search).startTime.getTime();
                                          break;
                                    }
                  } else {
                        if (destinationLeg.length == 1) {
                              if (arrivaltime[destinationLeg[0] - 1] != null)
                                    arrivetime =
arrivaltime[destinationLeg[0] - 1];
                        } else
                              arrivetime = arrive.endTime.getTime();
                  }
            }

            starttime = starttime(indexFirstLeg, request, leg);
            int capacityToFind = crequest - leg.assignedcontainers;

            //Riccardo: avoid going back
            ArrayList<String> bannedStops = new ArrayList<>();
            for(int i = 0; i < newitinerary.legs.size(); i++){
                  Leg l = newitinerary.legs.get(i);
                  if(l.from.name.equals(leg.from.name)) break;
                  bannedStops.add(l.from.name);
            }
            int nextLastToBan = 0;
            for(int item : lista) {
                  if(item > indexLastLeg) {
                        nextLastToBan = item;
                        break;
                  }
            }
            if(nextLastToBan>0) {

      bannedStops.add(newitinerary.legs.get(nextLastToBan).to.name);
```

```
                LOG.info(newitinerary.legs.get(nextLastToBan).to.name);
            }

            List<Itinerary> listfound =
srf.findOtherItineraryForCapacity(departure, arrival, starttime, null,
bannedStops,arrivetime, capacityToFind);
            Boolean yeswecan = false;
            if (listfound != null) {
                LOG.info("Listfound != null");
                for (Itinerary found : listfound) {
                    Boolean capacity = true;
                    for (Leg nleg : found.legs) {
                        if (!nleg.isTRUCKLeg()) {
                            if (nleg.capacity < capacityToFind) {
                                LOG.info("   New ininerary is not
good");

                                capacity = false;
                                break;
                            }
                        }
                        for (Leg check : newitinerary.legs) {
                            Boolean doublecheck;
                            doublecheck = checkleg(nleg, check);
                            if (!doublecheck) {
                                capacity = false;
                                break;
                            }
                        }
                        if (!capacity)
                            break;
                    }
                    if (capacity) {
                        int n = 0, m = found.legs.size();
                        if (newitinerary.endTime.before(found.endTime))
                            newitinerary.endTime = found.endTime;
                        for (Leg nleg : found.legs) {
                            nleg.assignedcontainers = capacityToFind;
                            nleg.destinationId = leg.destinationId;
                            nleg.previousDestinationId =
leg.previousDestinationId;

                            nleg.capacityId = "2";
```

```
                                        if (indexFirstLeg > 0 && n == 0
                                                        &&
(leg.destinationId.equals(leg.previousDestinationId)
                                                                        ||
leg.previousDestinationId.equals("0")
                                                                        ||
leg.destinationId.equals(newitinerary.legs.get(prev).destinationId))) {
                                                nleg.prevCapacityId =
Long.toString(newitinerary.legs.get(prev).getID());
                                        }
                                        if (n > 0)
                                                nleg.prevCapacityId =
Long.toString(found.legs.get(n - 1).getID());
                                        if (n < m - 1)
                                                nleg.nextCapacityId =
Long.toString(found.legs.get(n + 1).getID());
                                        if (indexLastLeg < ultimo && n == m - 1)
{
                                                nleg.nextCapacityId =
Long.toString(newitinerary.legs.get(nt).getID());
                                        }

      //refreshCostCo2legparallels(newitinerary, nleg, capacityToFind, true);
                                        newitinerary.legs.add(nleg);
                                        n++;
                                }
                                if (movetruck) {
                                        if
(newitinerary.legs.get(nt).isTRUCKLeg()) {
                                                if
(newitinerary.legs.get(nt).startTime.before(found.endTime)) {
                                                        long duration =
newitinerary.legs.get(nt).getDurationLong();

      newitinerary.legs.get(nt).startTime = found.endTime;

      newitinerary.legs.get(nt).setEndTime(found.endTime.getTimeInMillis() +
duration);
                                                }
                                        }
                                }
```

```
                                yeswecan = true;
                                break;
                        }
                }
        }
        if (yeswecan) {
                LOG.info("Found!");
                return indexLastLeg;
        } else {
                LOG.info("No good");
                if (nLegsOriginalItinerary > 2) {
                        if (indexLastLeg < ultimo) {
                                LOG.info("Try to another");
                                int ret =
searchotheritinerarymultiple(newitinerary, indexFirstLeg, nt,
nLegsOriginalItinerary,
                                                request, crequest, arrivaltime,
capacityRequest);
                                if (ret > 0) {
                                        return ret;
                                } else {
                                        LOG.info("searchotheritinerary non
found");
                                        return -1;
                                }
                        } else
                                return -1;
                } else
                        return -1;
        }
    }
```

Questo metodo è il **cuore** della “Capacity Mode”: dato un itinerario, la prima tratta che non soddisfa la richiesta viene passata come indice e il metodo stesso controlla se non ci sono tratte successive a questa che non soddisfano anch’esse i requisiti: a questo punto anziché cercare due tratte separate da mettere in parallelo, si cerca un **intero tratto** dal punto di partenza della prima tratta fino al punto di arrivo dell’ultima tratta che non rispetta i requisiti; ma di tratte che vanno verso le **stesse destinazioni**. Se non viene trovata nessuna tratta che soddisfa i requisiti il sistema cerca di “agganciarsi” all’itinerario al nodo successivo (anche

se la tratta che arriva a questo nodo soddisfa i risultati) fino al nodo destinazione (o l’ultimo nodo che rappresenta un insieme di destinazioni, che poi dopo questo punto le tratte si separano) in modo ricorsivo utilizzando il suddetto metodo. Prima di chiedere all’OTP tramite la variabile globale *SpecialRouteFinder* la **tratta aggiuntiva** bisogna settare la partenza e l’arrivo della tratta parallela, l’orario di partenza (che dipende dal punto dell’itinerario in cui la tratta si trova, se è la prima l’orario è quello settato dai requisiti, se no dipende dalla tratta precedente), l’orario di arrivo (come l’orario di partenza, bisogna capire se l’utente ha settato l’orario di arrivo in caso l’ultimo nodo da controllare sia una destinazione, ed eventualmente se la tratta successiva è di tipo truck trasliamo l’orario dei truck poiché la loro partenza può essere spostata dipendentemente dalla tratta successiva a quella dei truck). In più bisogna settare i **nodi** che questa tratta non deve considerare come ad esempio il nodo successivo nell’itinerario poiché non ha senso andare in una città per poi tornare indietro. Il metodo restituisce un numero negativo se non è stata trovata una tratta parallela e non è più possibile spostare il nodo di arrivo; invece restituisce un numero positivo ad indicare fino a quale tratta dell’itinerario originale il metodo ha trovato una tratta parallela.
Le tratte in parallelo vengono messe al fondo della lista e vengono anche visualizzate dopo le tratte originali.

2.4.1.9 checkprev

```
	private int checkprev(Itinerary newitinerary, int indexFirstLeg, Leg leg)
{
		StringTokenizer st = new StringTokenizer(leg.destinationId, "-");
		int prev = -1;
		List<Integer> list = new ArrayList<>();
		while (st.hasMoreElements()) {
			list.add(Integer.parseInt(st.nextElement().toString()));
		}
		int count = 0;
		for (Leg nleg : newitinerary.legs) {
			if (count < indexFirstLeg) {
				List<Integer> leglist = new ArrayList<>();
				StringTokenizer str = new
StringTokenizer(nleg.destinationId, "-");
				while (str.hasMoreElements()) {
```

```
	leglist.add(Integer.parseInt(str.nextElement().toString()));
                        }
                        Boolean ok = false;
                        for (int i = 0; i < list.size(); i++) {
                              for (int j = 0; j < leglist.size(); j++) {
                                    if (list.get(i).equals(leglist.get(j))) {
                                          ok = true;
                                    }
                              }
                              if (!ok)
                                    break;
                        }
                        if (ok)
                              prev = count;
                  }
                  count++;
            }
            return prev;
      }
```

Questo metodo serve per cercare l'indice della tratta precedente all'indice dato come parametro. La tratta precedente deve avere gli stessi nodi di destinazione della tratta selezionata.

2.4.1.10 getDestinationId

```
      private int[] getDestinationId(String destinationID) {
            String[] destination = destinationID == null ? null :
destinationID.split("-");
            int[] destinationid = new int[destination.length];
            for (int i = 0; i < destination.length; i++) {
                  destinationid[i] = Integer.parseInt(destination[i]);
            }
            return destinationid;
      }
```

getDestionationId serve per trasformare la stringa che contiene gli indici della destinazione in un array di interi che per ogni cella dell'array si trova un indice di destinazione.

2.4.1.11 merge2itinerary

```
/**   private void merge2itinerary(int capacityRequest, List<Itinerary>
itinerariesFinal) {
            Map<Itinerary, Integer> mapItineraryforMerge2 = new
ConcurrentHashMap<>();
            for (Map.Entry<Itinerary, Integer> itinerary :
mapItineraryMaxCapacity.entrySet()) {
                  mapItineraryforMerge2.put(itinerary.getKey(),
itinerary.getValue());
            }
            LOG.info("Start search 2 itineraries");
            for (Map.Entry<Itinerary, Integer> itinerary :
mapItineraryMaxCapacity.entrySet()) {
                  for (Map.Entry<Itinerary, Integer> itinerary2 :
mapItineraryforMerge2.entrySet()) {
                        if (!itinerary.getKey().equals(itinerary2.getKey())) {
                              if (itinerary.getValue() +
itinerary2.getValue() >= capacityRequest) {
                                    Boolean check = true;
                                    for (Leg leg1 : itinerary.getKey().legs)
{
                                    for (Leg leg2 : itinerary2.getKey().legs)
{
                                          check = checkleg(leg1,
leg2);
                                          if (!check)
                                                break;
                                    }
                                    if (!check)
                                          break;
                              }
                              if (check) {
                              Itinerary newItinerary = new Itinerary();

      newItinerary.cloneItinerary(itinerary.getKey(), true);
                              Calendar partenza1 =
newItinerary.startTime;
                                    Calendar partenza2 =
itinerary2.getKey().startTime;
```

```
                                        Calendar arrivo1 =
newItinerary.endTime;
                                        Calendar arrivo2 =
itinerary2.getKey().endTime;
                                        int boarding1 = 0;
                                        int boarding2 = 0;
                                        int boarding = 0;
                                        Boolean flag = false;
                                        for (Leg leg : newItinerary.legs)
{
                                              LOG.info("Itinerary: leg
form: " + leg.from.name + " to: " + leg.to.name);
                                        leg.assignedcontainers =
itinerary.getValue();

      refreshCostCo2leg(newItinerary, leg, false, capacityRequest);
                                              if (!flag) {
                                                    boarding1 =
leg.boarding;

                                                    flag = true;
                                              }
                                        }
                                        flag = false;
                                        for (Leg legfound :
itinerary2.getKey().legs) {
                                              LOG.info("Itinerary2: leg
form: " + legfound.from.name + " to: " + legfound.to.name);
                                              Leg leg = new Leg();
                                              leg.cloneLeg(legfound);
                                              leg.capacityId = "2";
                                              leg.assignedcontainers =
capacityRequest - itinerary.getValue();
                                              if (leg.isTRUCKLeg()) {

      leg.costforonecontainerTruck = leg.totalcost;
                                                    leg.totalcost =
leg.totalcost * leg.assignedcontainers;
                                                    leg.CO2 = leg.CO2 *
leg.assignedcontainers;
                                                    leg.costCO2 =
leg.costCO2 * leg.assignedcontainers;
```

```
                                                }
                                                if (!flag) {
                                                      boarding1 =
leg.boarding;
                                                      flag = true;
                                                }

      newItinerary.addLegFromAnotherItineraryForCapacity(leg);
                                          }
                                          if (partenza2.before(partenza1)) {
                                                newItinerary.startTime =
partenza2;
                                                boarding = boarding2;
                                          } else {
                                                boarding = boarding1;
                                          }
                                          if (arrivo2.getTimeInMillis() >
arrivo1.getTimeInMillis()) {
                                                newItinerary.endTime =
arrivo2;
                                          }
                                          newItinerary.duration =
((newItinerary.endTime.getTimeInMillis()
                                                      -
newItinerary.startTime.getTimeInMillis()) / 1000) + boarding;

      itinerariesFinal.add(newItinerary);
                                    }
                              }
                        }

                  }
                  mapItineraryforMerge2.remove(itinerary.getKey());
            }
      } **/
```

Questo metodo come si può notare è interamente **commentato** poiché, come spiegato in precedenza, una **specifica** (poi rimossa) era l’**unione** di due itinerari che non soddisfano pienamente i requisiti ma che uniti riescono a portare tutti i container a destinazione. Gli itinerari sono sempre nella mappa *mapItineraryMaxCapacity* dove vengono clonati in una mappa interna per non andare ad intaccare gli altri metodi. Se due itinerari soddisfano con

la somma i requisiti vengono inseriti in un unico itinerario, dopo vengono aggiornati i costi e le emissioni e il nuovo itinerario viene **aggiunto** alla lista *itinerariesFinal*.

2.4.1.12 refreshCostCo2leg

```
	/*private void refreshCostCo2leg(Itinerary itinerary, Leg leg, Boolean
flag, int capacityRequest) {
		if (leg.isTRUCKLeg()) {
			itinerary.CO2TRUCK -= leg.CO2;
			itinerary.totalcostTRUCK -= leg.totalcost;
			itinerary.totalcost -= leg.totalcost;
			itinerary.CO2 -= leg.CO2;
			itinerary.costCO2 -= leg.costCO2;
			leg.costforonecontainerTruck = leg.totalcost;
			leg.totalcost = leg.totalcost * leg.assignedcontainers;
			if (!flag) {
				leg.CO2 = (leg.CO2 / capacityRequest) *
leg.assignedcontainers;
				leg.costCO2 = (leg.costCO2 / capacityRequest) *
leg.assignedcontainers;
			}
			itinerary.CO2TRUCK += leg.CO2;
			itinerary.totalcostTRUCK += leg.totalcost;
			itinerary.totalcost += leg.totalcost;
			itinerary.CO2 += leg.CO2;
			itinerary.costCO2 += leg.costCO2;
		}
	}*/
```

Anche questo metodo è commentato poiché era stato pensato per il metodo *merge2itinerary*: serviva per aggiornare i **costi complessivi** dell'itinerario al momento delle aggiunte delle nuove tratte all'itinerario originale.

2.4.1.13 refreshCostCo2legparallels

```
private void refreshCostCo2legparallels(Itinerary itinerary, Leg leg, int
capacityRequest, Boolean flag) {
		if (leg.isTRUCKLeg()) {
			if (flag) {
				leg.costforonecontainerTruck = leg.totalcost;
```

```
                        leg.totalcost = leg.totalcost *
leg.assignedcontainers;
                        leg.CO2 = (leg.CO2 / capacityRequest) *
leg.assignedcontainers;
                        leg.costCO2 = (leg.costCO2 / capacityRequest) *
leg.assignedcontainers;
                  }
                  itinerary.CO2TRUCK += leg.CO2;
                  itinerary.totalcostTRUCK += leg.totalcost;
                  itinerary.totalcost += leg.totalcost;
                  itinerary.CO2 += leg.CO2;
                  itinerary.costCO2 += leg.costCO2;
            }
            if (leg.isTRAINLeg()) {
                  itinerary.CO2TRAIN += leg.CO2;
                  itinerary.totalcostTRAIN += leg.totalcost;
                  itinerary.totalcost += leg.totalcost;
                  itinerary.CO2 += leg.CO2;
                  itinerary.costCO2 += leg.costCO2;
            }
            if (leg.isSHIPLeg()) {
                  itinerary.CO2SHIP += leg.CO2;
                  itinerary.totalcostSHIP += leg.totalcost;
                  itinerary.totalcost += leg.totalcost;
                  itinerary.CO2 += leg.CO2;
                  itinerary.costCO2 += leg.costCO2;
            }
      }
```

Questo è il metodo per **aggiornare i costi e le emissioni** dell’itinerario, diviso per tipologia di trasporto e in caso di truck ripartizionate le emissioni in base al numero di truck previsto.

2.4.1.14 starttime

```
private Date starttime(int indexFirstLeg, RoutingRequest request, Leg leg) {
            if (indexFirstLeg == 0)
                  return request.getDateTime();
            else
                  return leg.startTime.getTime();
      }
```

Questo metodo restituisce un oggetto di tipo *Date*: se l'indice è della prima tratta restituisce l'orario di partenza della richiesta; al contrario restituisce l'orario di partenza della tratta stessa.

2.4.1.15 checkleg

```
private Boolean checkleg(Leg nleg, Leg check) {
		if (nleg.mode.equals(check.mode)) {
			if (nleg.from.name.equals(check.from.name)) {
				if (nleg.to.name.equals(check.to.name)) {
					if (nleg.startTime.equals(check.startTime)) {
						if (nleg.endTime.equals(check.endTime)) {
							return false;
						}
					}
				}
			}
			return true;
	}
```

Questo metodo controlla che le due tratte passate per parametro non siano la stessa **identica** tratta: confronta il tipo di trasporto, il nodo di partenza, il nodo di arrivo, l'orario di partenza e l'orario di arrivo. Se la tratta è la stessa il metodo restituisce un booleano con il valore falso, nel caso contrario un booleano con il valore vero.

2.4.1.16 refreshTotalCost

```
private void refreshTotalCost(List<Itinerary> itinerariesFinal) {
		for (Itinerary it : itinerariesFinal) {
			it.CO2 = (double) 0;
			it.CO2SHIP = (double) 0;
			it.CO2TRAIN = (double) 0;
			it.CO2TRUCK = (double) 0;
			it.costCO2 = (double) 0;
			it.totalcost = (double) 0;
			it.totalcostSHIP = (double) 0;
```

```
                    it.totalcostTRAIN = (double) 0;
                    it.totalcostTRUCK = (double) 0;
                    for (Leg leg : it.legs) {
                          refreshCostCo2legparallels(it, leg, 0, false);
                    }
              }
        }
}
```

Questo metodo viene eseguito alla fine per **aggiornare** tutti i costi e le emissioni di tutti gli itinerari: prima vengono settati a 0 tutte le variabili dell’itinerario riguardanti costi e emissioni, e poi per ogni tratta di ogni itinerario viene lanciato il metodo *refreshCostCo2legparallels* già analizzato in precedenza.

2.5 Conclusioni analisi codice

Come si può notare molti metodi sono stati **modificati** più volte in base alle specifiche della “Capacity Mode”, altri non sono stati più utilizzati. In principio la “Capacity Mode” funzionava solo per la modalità single destination: successivamente è stata completamente rivista e modificata per funzionare anche in modalità multiple destination: ciò ha comportato ore di controlli, debug e **ottimizzazione** del codice appena analizzato

.

3 Analisi dei risultati

Dopo aver illustrato il funzionamento tecnico della “Capacity Mode” al livello di codice Java ora vorrei illustrare **i risultati** della modalità con le stesse tratte illustrate nel primo capitolo.

3.1 Esempio single destination mode

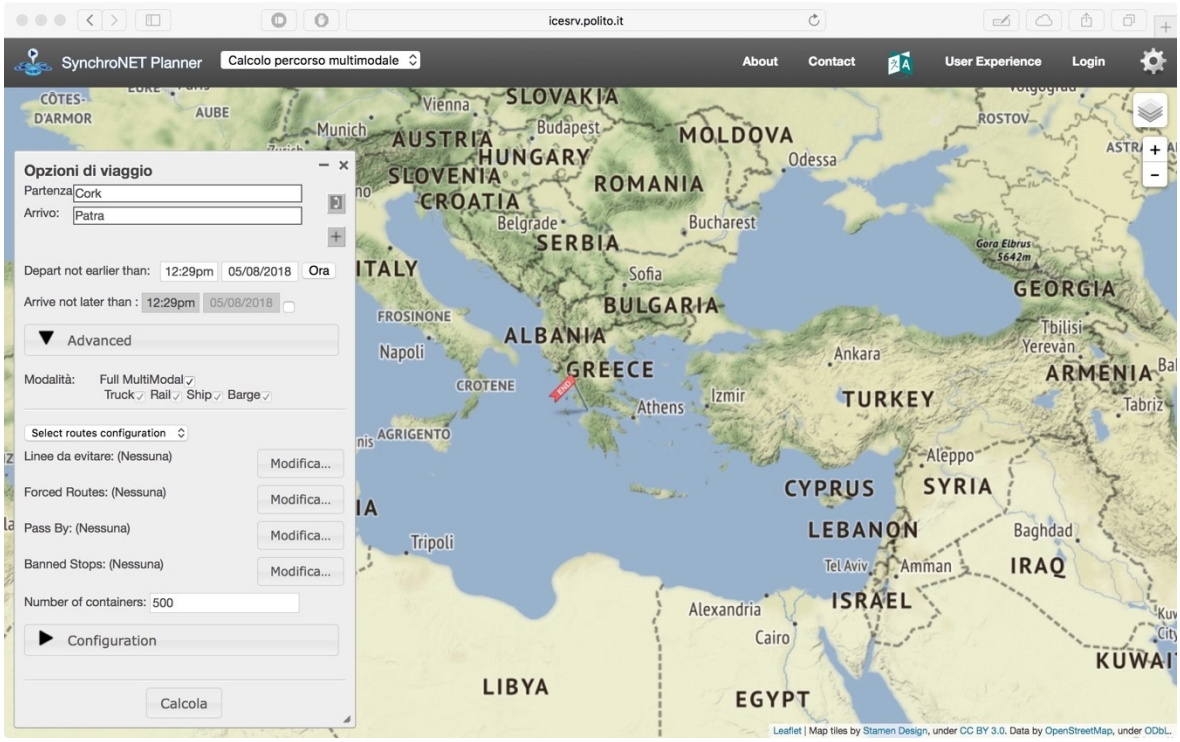
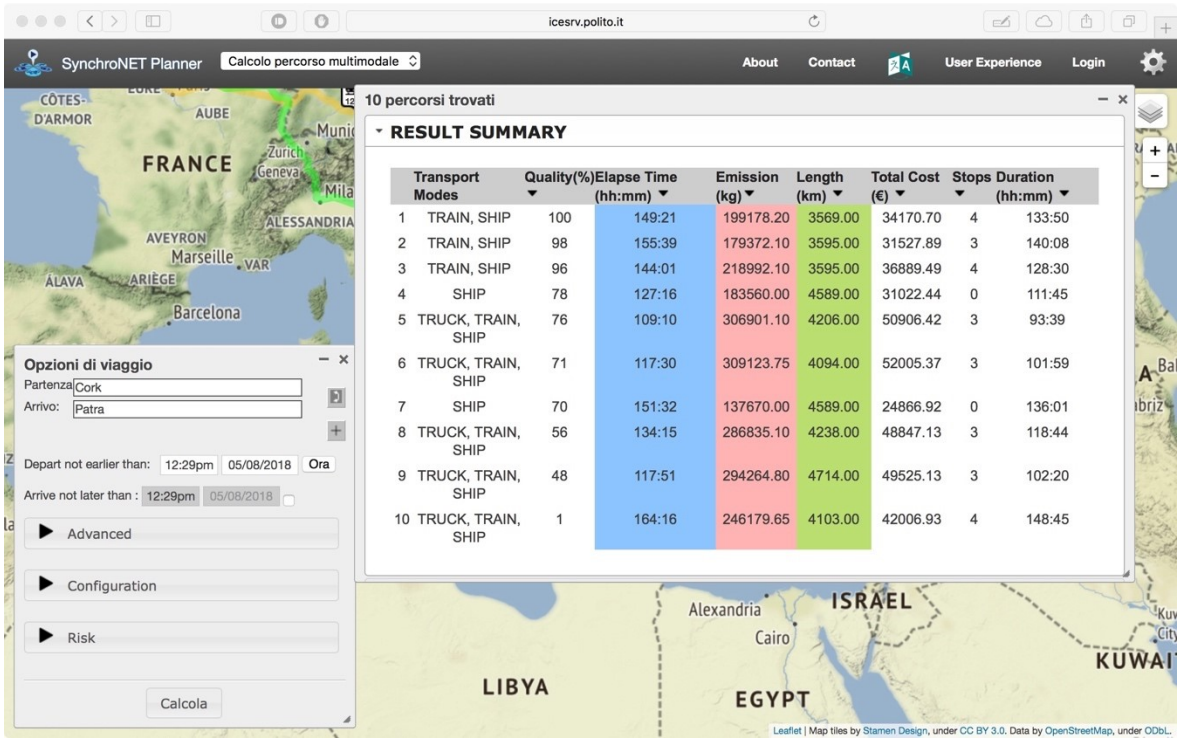


Figura 16 – Cork Patrasso con 500 container

Come si può notare dalla figura 16 nell’interfaccia grafica possiamo trovare una cella di input dove inserire **il numero di container** inviare alla destinazione: in questo caso si sta chiedendo al simulatore di trovare degli itinerari che vadano da Cork a Patrasso per il trasporto di 500 container.



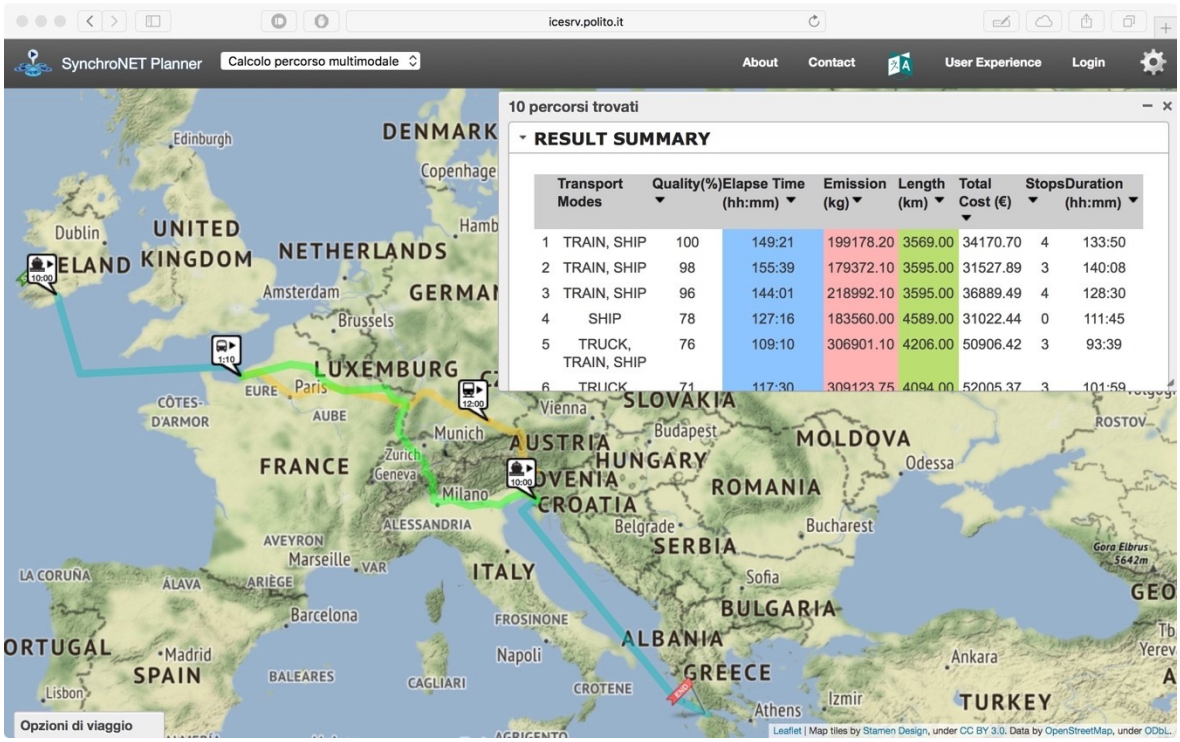


Figura 17a e 17b – Risultati della ricerca Cork Patrasso con 500 container

Come si può notare dalla figura 17b già la mappa può far capire come la “Capacity Mode” lavora: solo graficamente si riesce a intuire che a Le Havre i container si **separano** per poi riunirsi a Trieste; lungo questa tratta **alcuni container** andranno in **treno** e gli **altri** tramite **truck**. Inoltre, il software è riuscito comunque a garantire 10 itinerari che vanno dalla partenza alla destinazione trasportando tutti i container richiesti.
Come si può notare la “Capacity Mode” non modifica l’interfaccia grafica ma aggiunge solo rotte parallele per il trasporto di tutti i container.

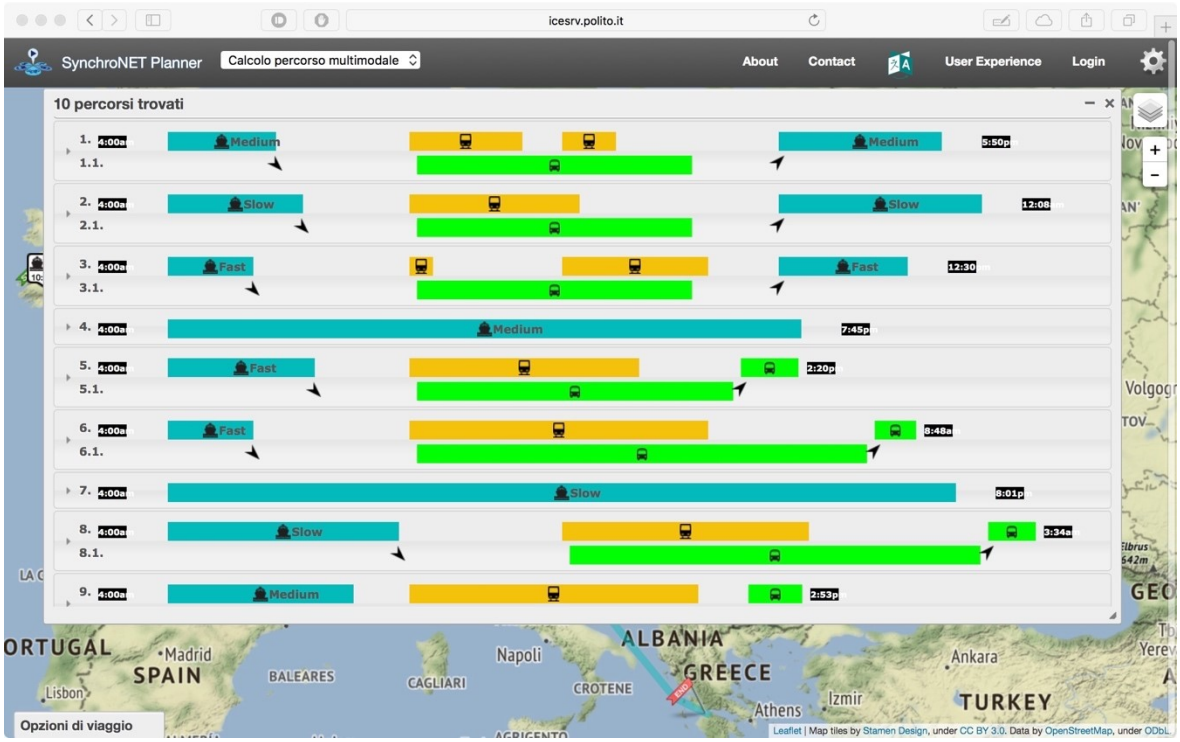
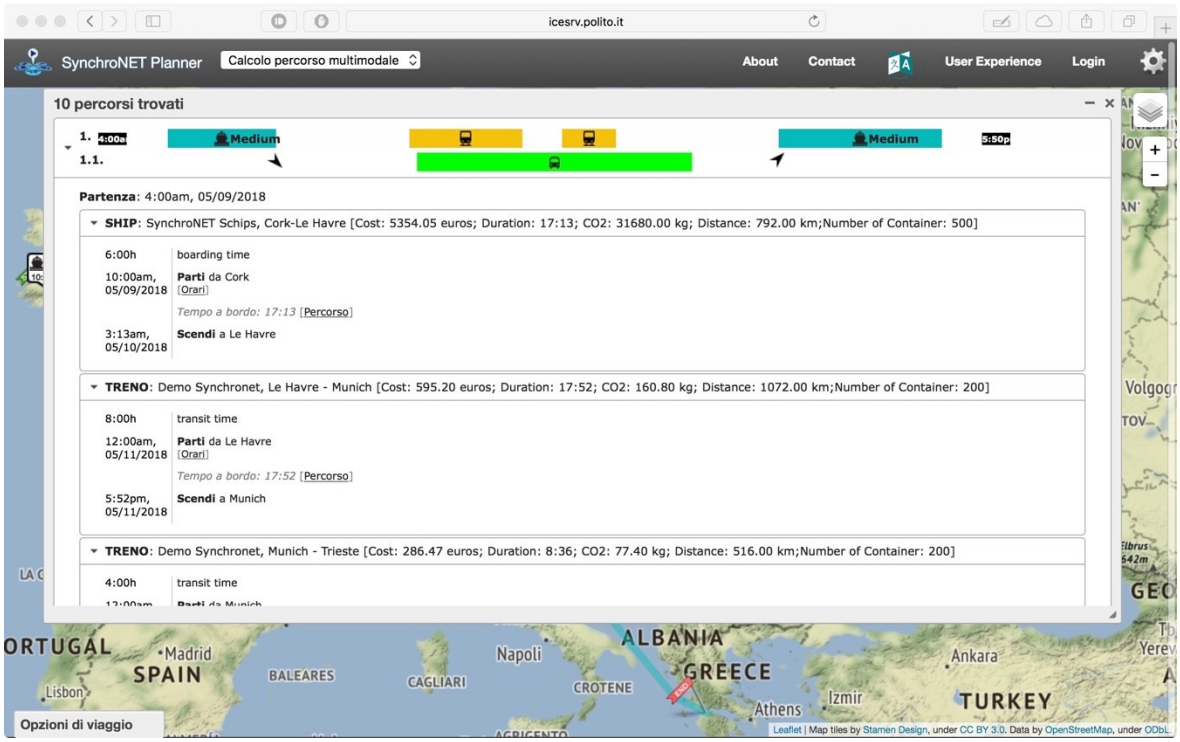


Figura 18 – Visualizzazione dei 10 itinerari Cork Patrasso da 500 container

Come si può notare dalla figura 18 graficamente si può capire l’andamento dei container tramite le **frecce** che indicano il punto di biforcazione dell’itinerario e delle tratte parallele aggiunte tramite la “Capacity Mode”.
Se si osserva attentamente la figura si può notare che l’itinerario 4 e 7 sono itinerari formati da una sola tratta in nave che parte da Cork e arriva a Patrasso, ma non garantisce il miglior risultato a livello di qualità, che viene garantito dagli altri itinerari.
Ora vorrei analizzare il primo itinerario trovato:



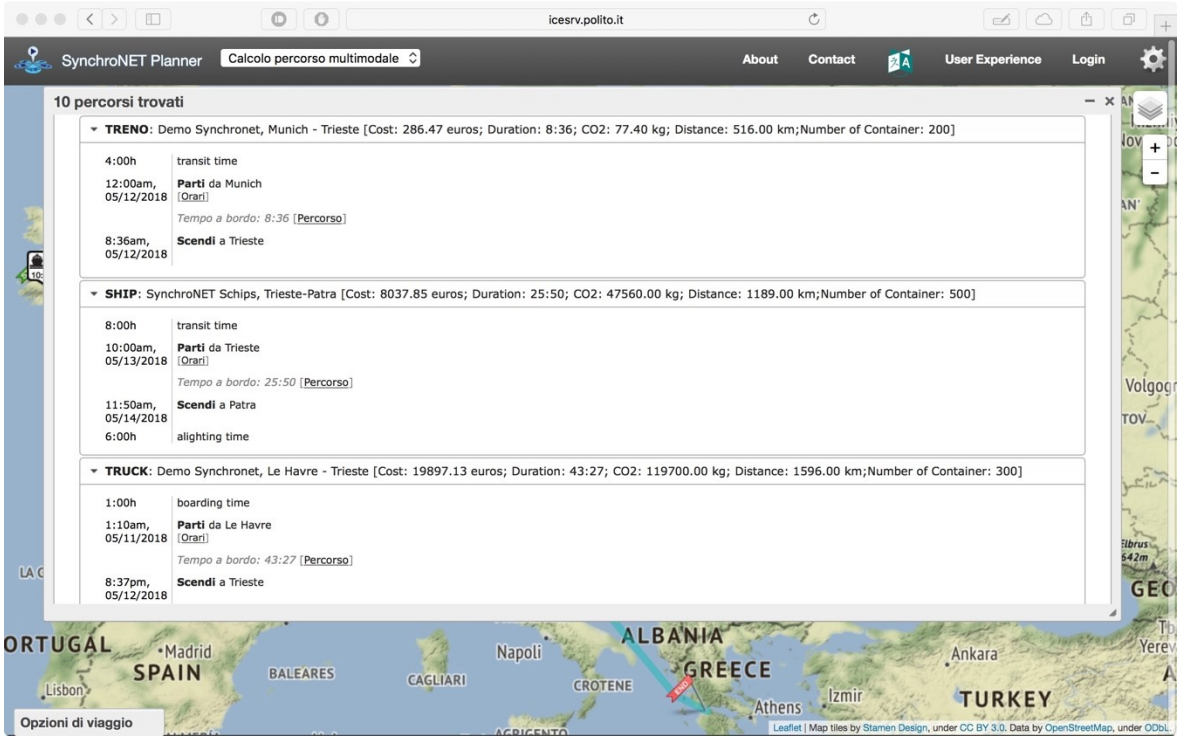


Figura 19a e 19b – Analisi del primo itinerario Cork Patrasso da 500 container

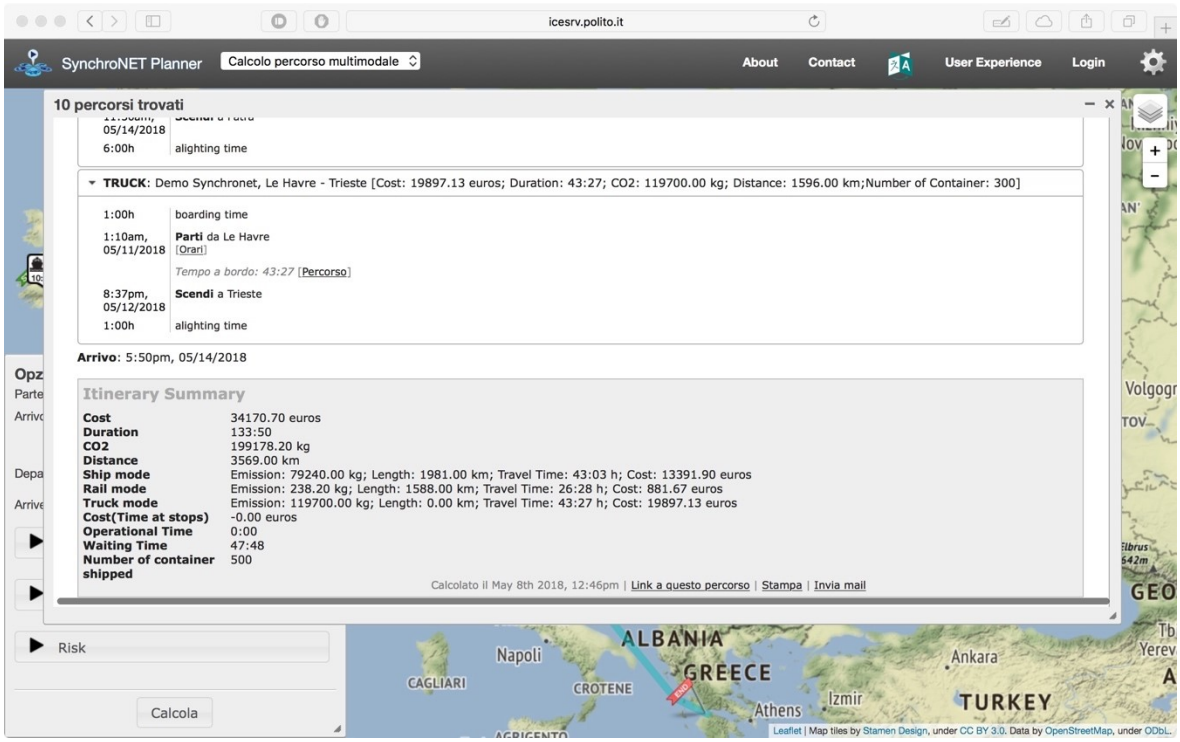


Figura 19c – Analisi del primo itinerario Cork Patrasso da 500 container

Nelle figure 19a, b e c possiamo analizzare il risultato ottenuto dal primo itinerario: in questo caso l'itinerario è composto da cinque tratte: Cork – Le Havre in nave; Le Havre – Monaco di Baviera in treno; Monaco di Baviera – Trieste sempre in treno; Le Havre – Trieste in truck e per concludere Trieste – Patrasso in nave. Entrando nel dettaglio della prima tratta Cork – Le Havre: l'output ci dice che la tratta è in nave; il costo della singola tratta è di 5354,05 euro; della durata di 17 ore e 13 minuti; 31680,00 kg di CO2 emessi e della distanza di 792 km. In più ci viene fornito che la partenza da Cork è alle 10:00 con l'orario di imbarco alle 6:00 del 09/05/2018 con l'arrivo a Le Havre alle 3:13 del 10/05/2018. Inoltre, ci viene segnalato che questa tratta trasporta 500 container.

La seconda tratta è Le Havre – Monaco di Baviera: la tratta è in treno; il costo della singola tratta è di 595,20 euro; della durata di 17 ore e 52 minuti; 160,80 kg di CO2 emessi e della distanza di 1072 km. In più ci viene fornito che la partenza da Le Havre è alle 12:00 del 11/05/2018 con l'arrivo a Monaco di Baviera alle 17:52 del 11/05/2018. I container trasportati da questa tratta sono 200.

La terza tratta è Monaco di Baviera – Trieste: la tratta è in treno; il costo della singola tratta è di 286,47 euro; della durata di 8 ore e 36 minuti; 77,40 kg di CO2 emessi e della distanza di 516 km. In più ci viene fornito che la partenza da Monaco di Baviera è alle 00:00 del 12/05/2018 con l'arrivo a Trieste alle 8:36 del 12/05/2018. I container trasportati sono 200.

La quarta tratta è Trieste – Patrasso: la tratta è in nave; il costo della singola tratta è di 8037,85 euro; della durata di 25 ore e 50 minuti; 47560,00 kg di CO2 emessi e della distanza di 1189 km. In più ci viene fornito che la partenza da Trieste è alle 10:00 del 13/05/2018 con l'arrivo a Patrasso alle 11:50 del 14/05/2018. I container trasportati sono 500.

Manca solo la tratta di completamento dei container: Le Havre – Trieste: la tratta è tramite truck; il costo della singola tratta è di 19897,13 euro; della durata di 47 ore e 27 minuti; 119700 kg di CO2 emessi e della distanza di 1596 km. In più ci viene fornito che la partenza da Le Havre è alle 1:10 del 11/05/2018 con l'arrivo a Trieste alle 20:37 del 12/05/2018. I container trasportati sono 300.

In fine dopo l'ultima tratta troviamo il **sommario** dell'itinerario con le solite informazioni di riepilogo già analizzate nel primo capitolo: l'unica aggiunta è sul **numero complessivo di container** spedito che in questo caso è 500.

3.2 Esempio multi destination mode

Dopo aver visto un esempio della modalità single destination ora vedremo un'applicazione pratica della "Capacity Mode" nel caso di multi destination.

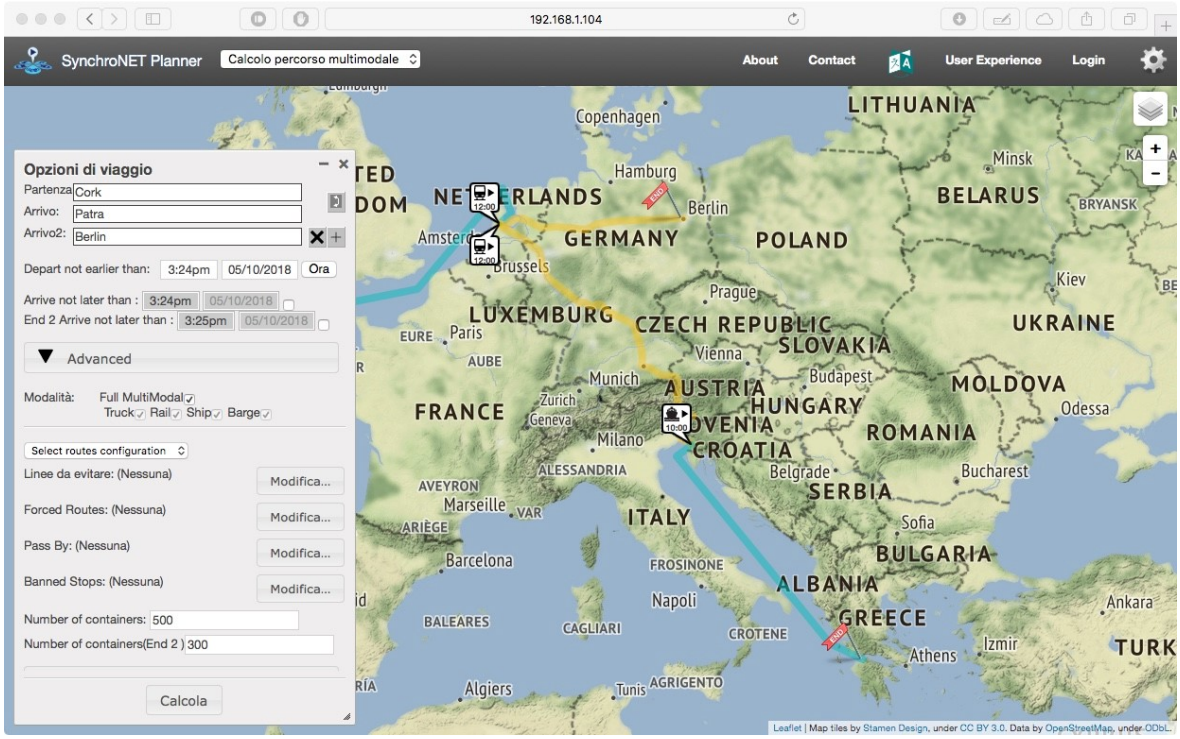


Figura 20 – Cork Patrasso con 500 container e Berlino con 300 container.

Come si può notare nella figura 20 al momento dell'aggiunta di più di una destinazione l'interfaccia grafica mostra più campi di inserimento per il numero di container da inviare per ogni destinazione aggiunta. In questo caso vedremo gli itinerari con partenza Cork e destinazioni Patrasso e Berlino con 500 **container** da spedire verso Patrasso e 300 verso la capitale tedesca.

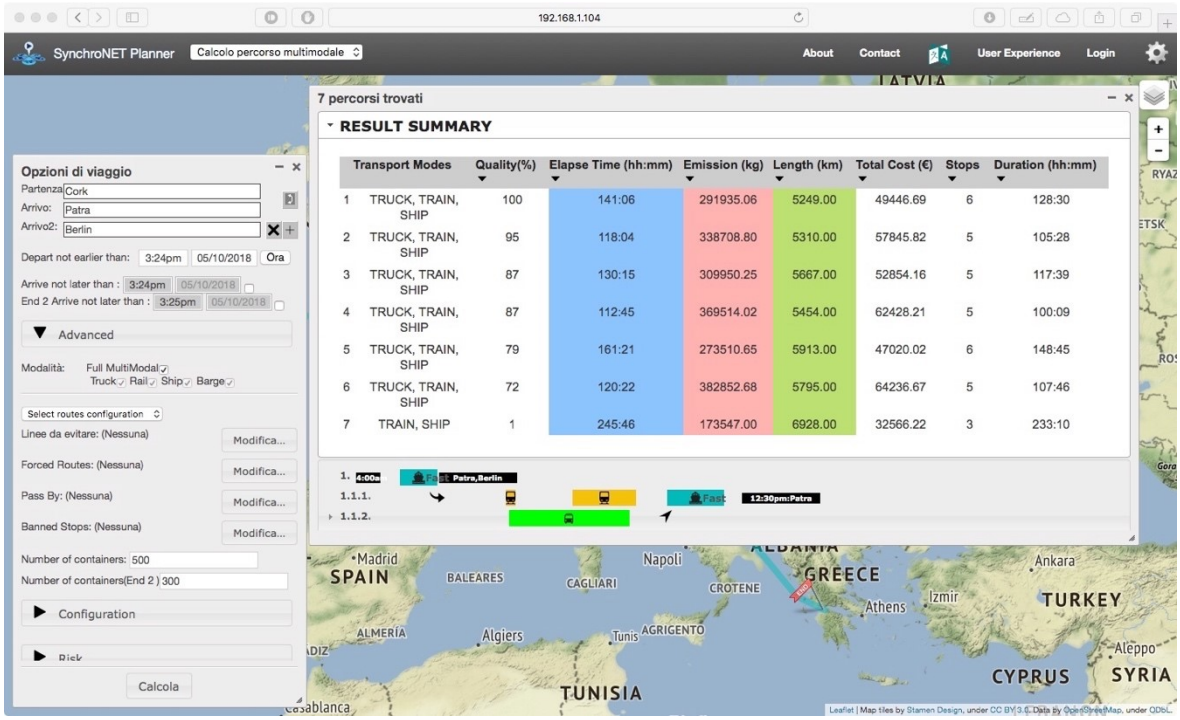


Figura 21a - Risultati Cork Patrasso Berlino

La figura 21a riassume l'output che il simulatore ha restituito al client: sono stati trovati 7 itinerari che da Cork portano 500 container a Patrasso, 300 container a Berlino con almeno una tratta in comune tra le due destinazioni. Nelle tratte in comune i container trasportati sono la **somma** di tutti i numeri di **container** verso le destinazioni di tale tratta.

Nelle figure 21b e 21c si possono notare graficamente i 7 percorsi trovati: la grafica riprende sia quella vista nel primo capitolo che quella vista sopra nella single destination "Capacity" mode. La freccia con la coda indica la divisione di destinazione, ergo la multiple destination; invece l'altra freccia indica il rientro da parte di una tratta parallela per il trasporto dei container rimanenti. Per facilitare la lettura oltre le frecce troviamo dei numeri affianco ad ogni riga: il primo numero indica il numero di itinerario, il secondo indica il numero della destinazione (nel caso del primo itinerario 1 sta per Patrasso e 2 per Berlino) e infine il terzo numero indica la divisione interna data dalla "Capacity Mode".

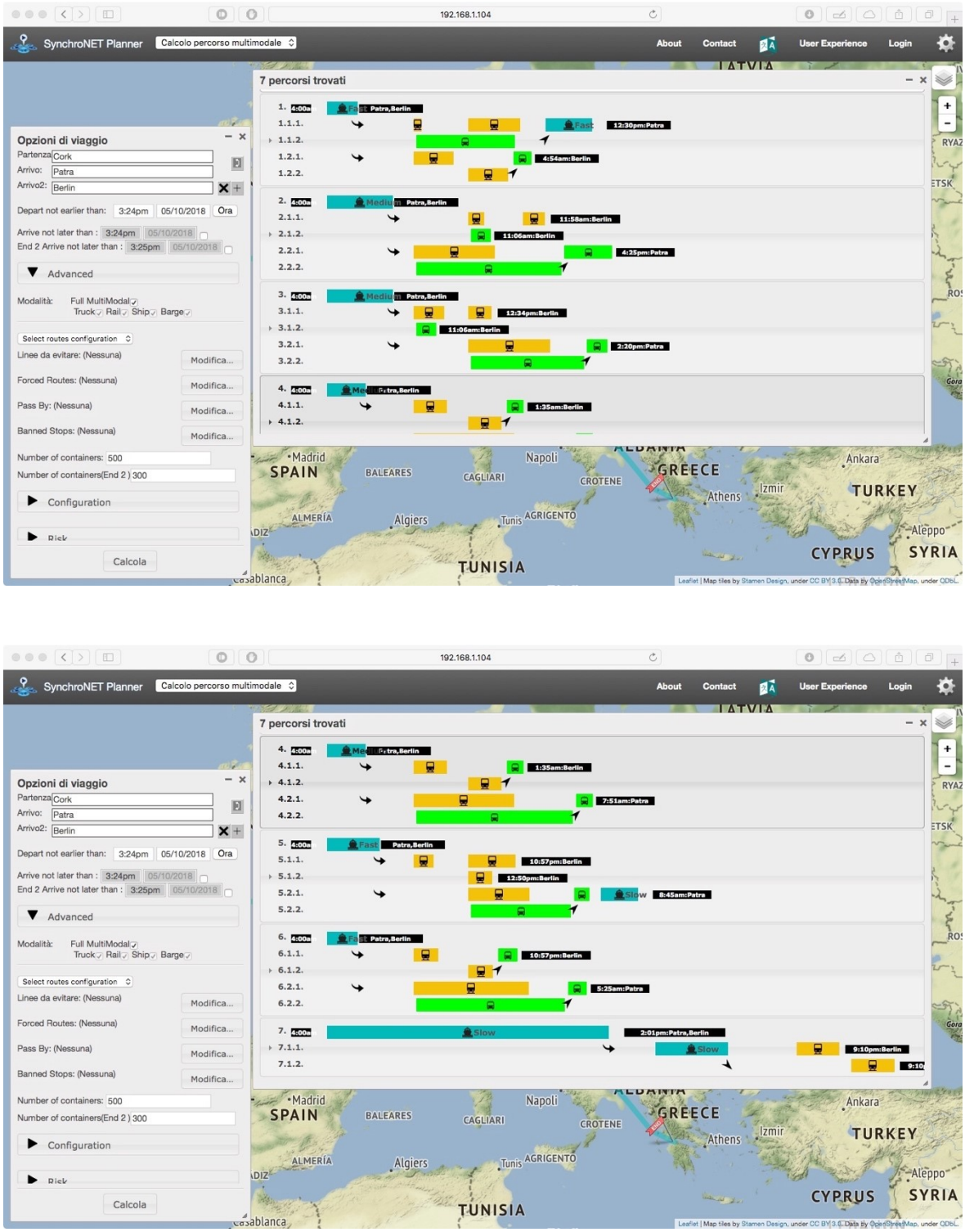
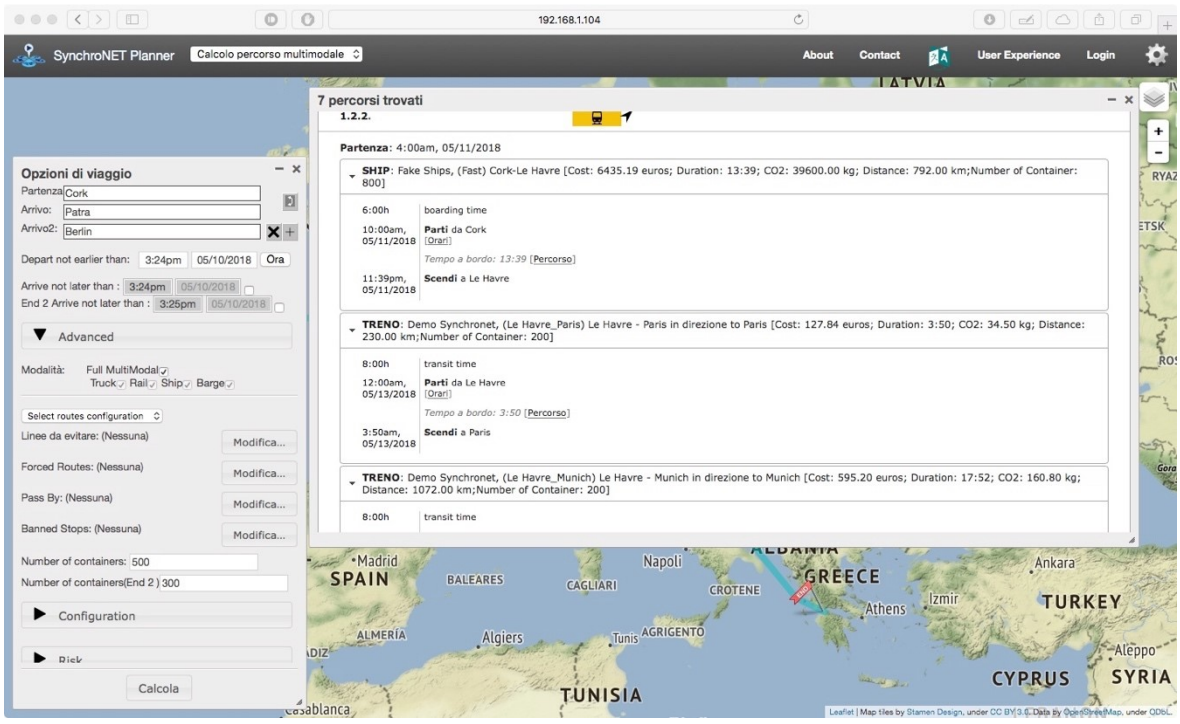


Figure 21b e 21c – Visualizzazione dei 7 percorsi trovati Cork Patrasso Berlino

Nella figura 22a, b, c, d ed e viene analizzato il primo percorso trovato dal simulatore per il percorso Cork Patrasso Berlino. Il percorso ha solo **una tratta in comune** che è la prima da Cork a Le Havre che trasporta **800 container**: le due destinazioni si separano a Le Havre dove si **dividono** nuovamente per "colpa" del numero di container; partono due tratte verso Patrasso da Le Havre e altre due tratte verso Berlino.



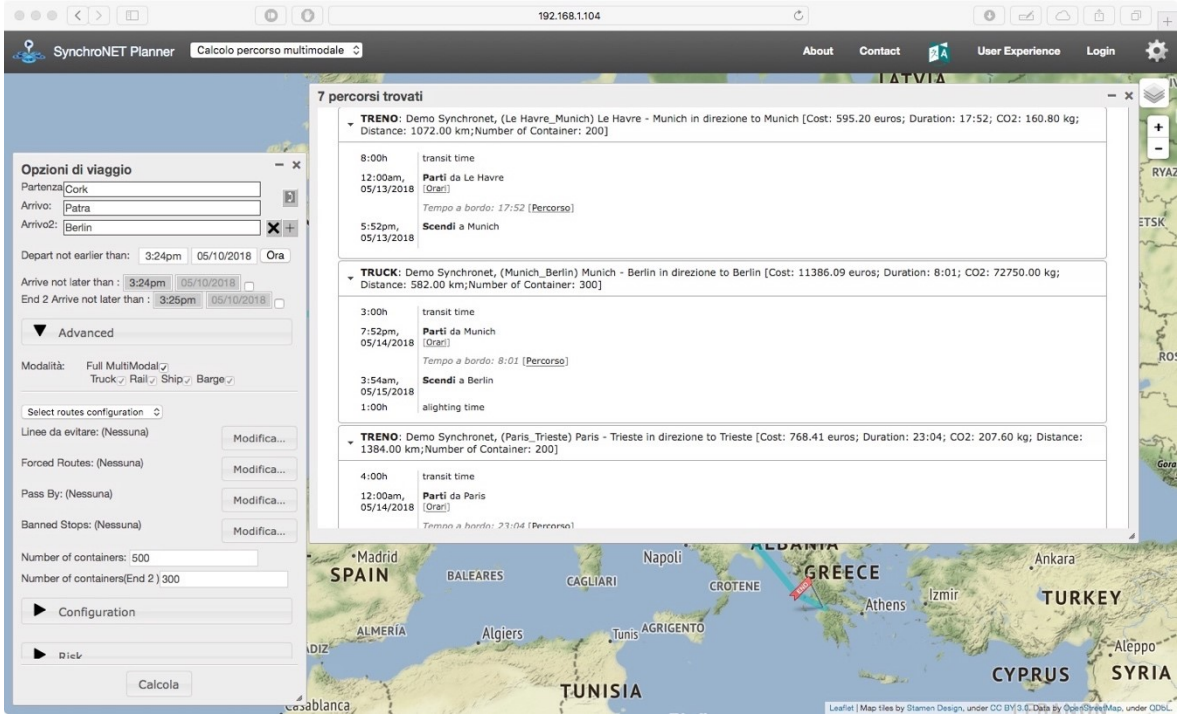


Figure 22a e 22 b – Primo itinerario Cork Patrasso Berlino

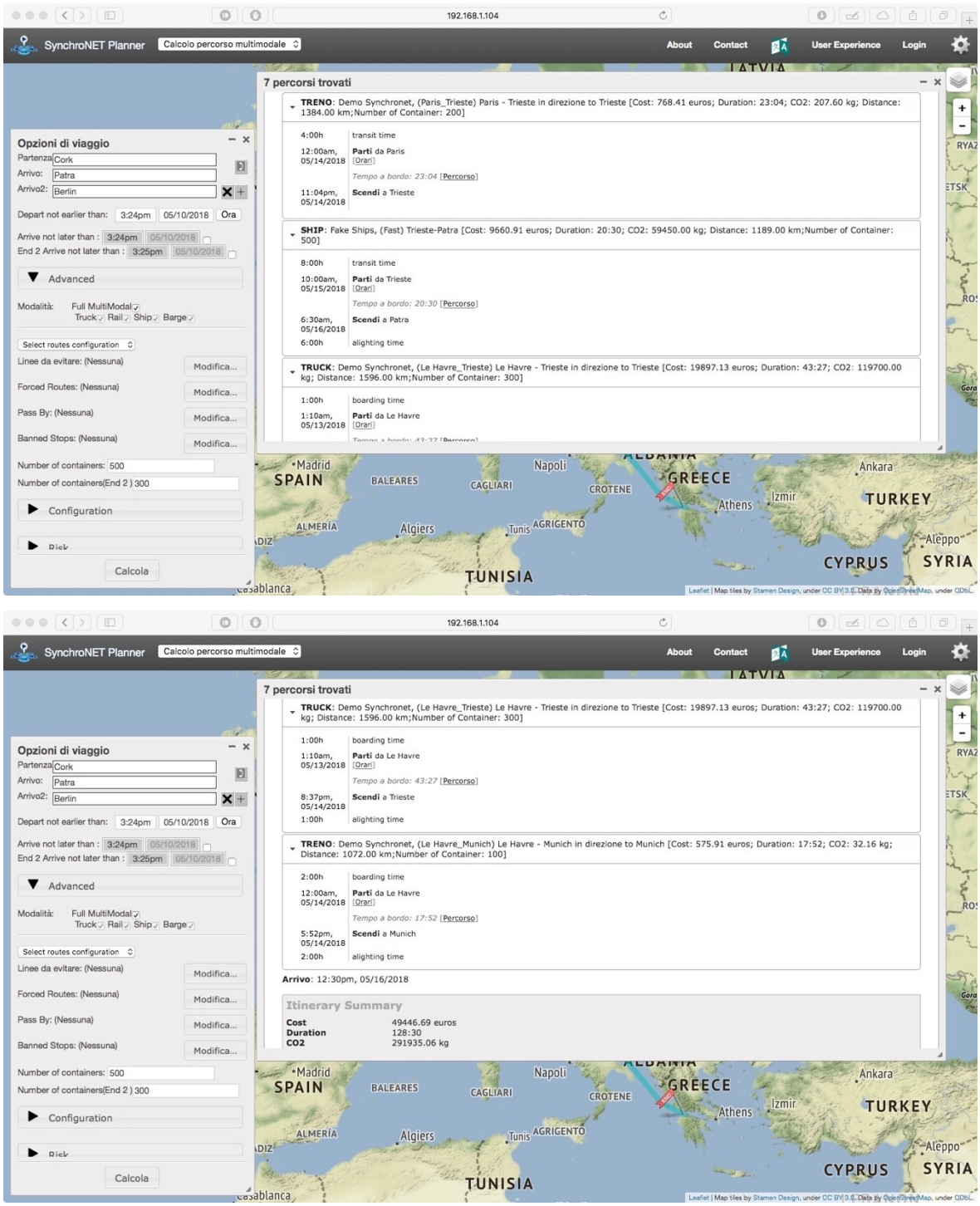


Figura 22 c e 22 d – Primo itinerario Cork Patrasso Berlino

La prima tratta è Cork – Le Havre: la tratta è in nave in direzione Patrasso e Berlino; il costo della singola tratta è di 6435,19 euro; della durata di 13 ore e 39 minuti; 39600 kg di CO2 emessi e della distanza di 792 km. Vengono trasportati 800 container in questa tratta. In più ci viene fornito che la partenza da Cork è alle 10:00 del 11/05/2018 con l'imbarco alle 04:00 e l'arrivo a Le Havre alle 22:39 del 11/05/2018.

La seconda tratta è Le Havre – Parigi: la tratta è in treno in direzione Patrasso; il costo della singola tratta è di 127,84 euro; della durata di 3 ore e 50 minuti; 34,50 kg di CO2 emessi e

della distanza di 230 km. Vengono trasportati 200 container in questa tratta. In più ci viene fornito che la partenza da Le Havre è alle 00:00 del 13/05/2018 e l'arrivo a Parigi alle 3:50 del 13/05/2018.

La terza tratta è Le Havre – Monaco di Baviera: la tratta è in treno in direzione Berlino; il costo della singola tratta è di 595,20 euro; della durata di 17 ore e 52 minuti; 160,80 kg di CO2 emessi e della distanza di 1072 km. Vengono trasportati 200 container in questa tratta. In più ci viene fornito che la partenza da Le Havre è alle 00:00 del 13/05/2018 e l'arrivo a Monaco di Baviera alle 17:52 del 13/05/2018.

La quarta tratta è Monaco di Baviera – Berlino: la tratta è tramite truck in direzione Berlino; il costo della singola tratta è di 11386,09 euro; della durata di 8 ore e 1 minuti; 72750 kg di CO2 emessi e della distanza di 582 km. Vengono trasportati 300 container in questa tratta. In più ci viene fornito che la partenza da Monaco di Baviera è alle 19:52 del 14/05/2018 e l'arrivo a Berlino alle 3:54 del 15/05/2018.

La quinta tratta è Parigi – Trieste: la tratta è in treno in direzione Patrasso; il costo della singola tratta è di 768,41 euro; della durata di 23 ore e 4 minuti; 207,60 kg di CO2 emessi e della distanza di 1384 km. Vengono trasportati 200 container in questa tratta. In più ci viene fornito che la partenza da Parigi è alle 00:00 del 14/05/2018 e l'arrivo a Trieste alle 23:04 del 14/05/2018.

La sesta tratta è Trieste – Patrasso: la tratta è in nave in direzione Patrasso; il costo della singola tratta è di 9660,91 euro; della durata di 20 ore e 30 minuti; 59450 kg di CO2 emessi e della distanza di 1189 km. Vengono trasportati 500 container in questa tratta. In più ci viene fornito che la partenza da Trieste è alle 10:00 del 15/05/2018 e l'arrivo a Patrasso alle 6:30 del 16/05/2018.

La settima tratta è Le Havre – Trieste: la tratta è tramite truck in direzione Patrasso; il costo della singola tratta è di 19897,13 euro; della durata di 43 ore e 27 minuti; 119700 kg di CO2 emessi e della distanza di 1596 km. Vengono trasportati 300 container in questa tratta. In più ci viene fornito che la partenza da Le Havre è alle 1:10 del 13/05/2018 e l'arrivo a Trieste alle 20:37 del 14/05/2018. Questa tratta compensa i container mancanti da Le Havre a Trieste, dove l'itinerario principale passa da Parigi tramite treno ma questi container vanno da Le Havre direttamente a Trieste.

L'ottava e ultima tratta è Le Havre – Monaco di Baviera: la tratta è in treno in direzione Berlino; il costo della singola tratta è di 575,91 euro; della durata di 17 ore e 52 minuti; 32,16 kg di CO2 emessi e della distanza di 1072 km. Vengono trasportati 100 container in questa tratta. In più ci viene fornito che la partenza da Le Havre è alle 00:00 del 14/05/2018 e

l'arrivo a Monaco di Baviera alle 17:52 del 14/05/2018. Questa tratta compensa i container mancanti da Le Havre a Monaco di Baviera ed è esattamente lo stesso treno ma del giorno dopo.

Alla fine, troviamo sempre il sommario finale con i riassunti dei costi e delle emissioni dell'itinerario.

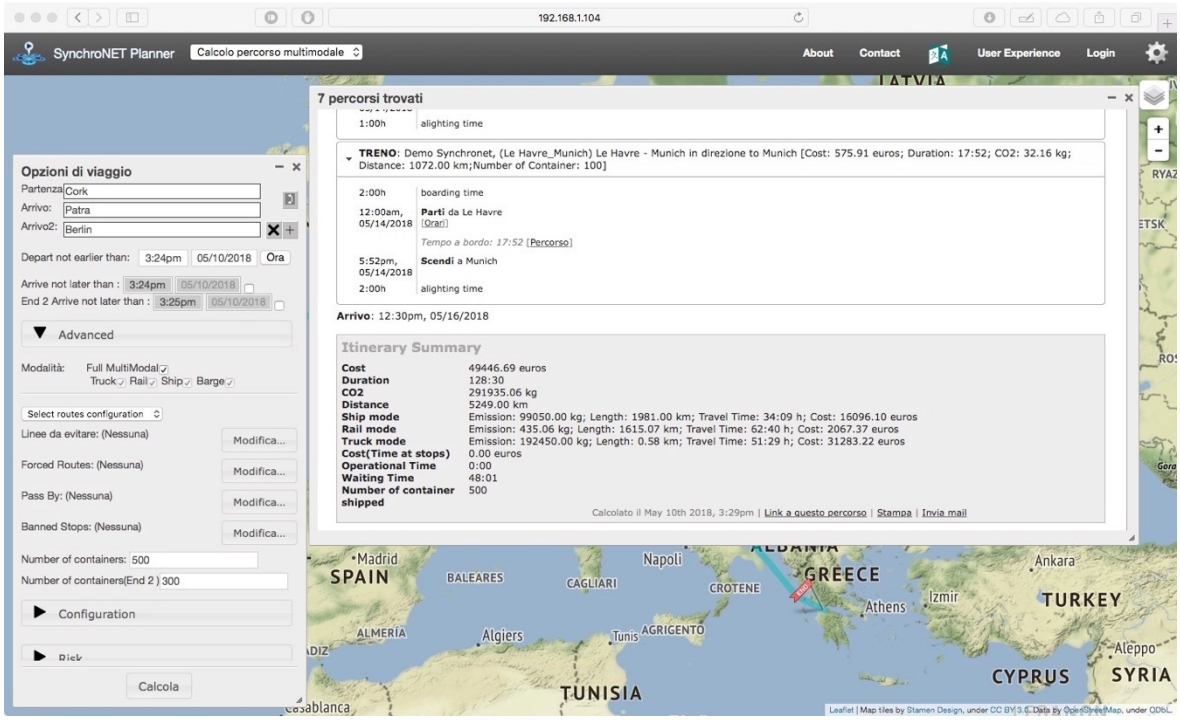


Figura 22e – Primo itinerario Cork Patrasso Berlino

3.3 Efficienza delle ricerche

Data l'introduzione della "Capacity Mode" all'interno dell'OTP si è voluto verificare l'impatto sulle prestazioni della suddetta modalità nell'utilizzo del simulatore.

In primo luogo, si è calcolato il tempo di risposta senza la modalità attiva in single mode (tratta Cork-Patrasso) e in multi destination mode (Cork-Patrasso-Berlino) 20 volte, per entrambe le modalità; successivamente si è ripetuto lo stesso test attivando la "Capacity Mode" inviando 500 container verso Patrasso e 200 container verso Berlino (solo nella multi destination) sempre per 20 volte.

Calcolando la media per le singole modalità e poi successivamente confrontandole con le stesse tratte ma con la "Capacity Mode" attiva si può concludere che c'è un aumento di circa 3 secondi nelle richieste con la richiesta di containers (Appendice – Tabella 1).

4 Conclusioni

L'argomento trattato in questa tesi aggiunge una **caratteristica** aggiuntiva al progetto in sé, rende possibile l'utilizzo in maniera più concreta e realistica; infatti ora è possibile:

- inserire un numero di container da spedire alle singole destinazioni;
- sostituire una singola tratta che non soddisfa la richiesta iniziale;
- verificare se un intero itinerario abbia o no i criteri capacitivi minimi per il trasporto di tutti i container;
- verificare se anche la tratta successiva a quella che non soddisfa i requisiti rispetti anch'essa la richiesta iniziale;
- evitare che la tratta "in parallelo" torni indietro;
- nel caso dei truck, spostare l'orario di partenza se la tratta "in parallelo" dovrebbe arrivare dopo l'orario di default, ma in tempo per la tratta successiva;
- visualizzare graficamente le tratte "in parallelo" e dove si ricongiungono (nel caso avenisse);
- rispettare gli orari di partenza e arrivo richiesti in fase di input;
- e inoltre tutto ciò funziona anche in modalità Multiple destination con lo stesso e identico algoritmo.

Inoltre, una volta popolati concretamente i GTFS crea una **simulazione reale** della situazione mondiale dei trasporti merci: il simulatore terrà conto delle capacità di tutte le rotte in tempo reale facendo in modo che chi utilizza il simulatore può avere un vero prospetto. L'OpenTripPlanner, in versione demo, si può utilizzare all'indirizzo: http://icesrv.polito.it:8080.

Tuttavia, è importante tenere presente che questa implementazione si è concentrata esclusivamente sui **risultati svolti** dal OTP senza modificare il processo esistente. Una raccomandazione per ulteriori ottimizzazioni future potrebbe essere quella di anticipare la "Capacity Mode" già a monte senza aspettare gli itinerari calcolati dall'OTP per poi modificarli cercando di diminuire l'attuale ritardo di circa 3 secondi.

Bibliografia

R. Giusti, D. Manerba, G. Perboli, R. Tadei, S. Yuan. A New Open-source System for Strategic Freight Logistics Planning: the SYNCHRO-NET Optimization Tools. *Transportation Research Procedia (forthcoming).* EURO mini conference on "Advances in Freight Transportation and Logistics", 7-9 March 2018. Padova, Italy.

D. Holfeld, C. Iorfida, M. Koya, D. Manerba, J. Stephens, R. Tadei, F. Werner. SYNCHRO-NET: a powerful and innovative synchro-modal supply chain eco-NET. *Proceedings of TRA2018, Transport Research Arena*. April 16-19, 2018. Vienna, Austria. (2018a)

D. Holfeld, A. Simroth, Y. Li, D. Manerba, R. Tadei. Risk Analysis for synchro-modal freight transportation: the SYNCHRO-NET approach. *Odysseus2018 - 7th International Workshop on Freight Transportation and Logistics*, June 3-8, 2018. Cagliari, Italy. (2018b)

Mes, M. R. K., Iacob, M.-E., 2016. Synchromodal Transport Planning at a Logistics Service Provider. In: Zijm, H., Klumpp, M., Clausen, U. (eds.), Logistics and Supply Chain Innovation, Lecture Notes in Logistics, pp. 23-36. Springer.

Perboli, G., Musso, S., Rosano, M., Tadei, R., Godel, M., 2017. Synchro-Modality and Slow Steaming: New Business Perspectives in Freight Transportation. Sustainability 2017, 9 (10), 1843; doi:10.3390/su9101843.

Simroth, A., Holfeld, D., Tadei, R., 2016. Risk Analysis for a Synchro-modal supply chain. 28th European Conference on Operational Research (EURO2016). Poznan, Poland.

Appendice

	No Capacity Mode		Si Capacity Mode	
	Cork-Patra	Cork-Patra-Berlin	Cork-Patra	Cork-Patra-Berlin
	Single Destination	Multi Destination	Single Destination	Multi Destination
1	861	2380	4030	6180
2	858	1490	4070	6790
3	894	2590	4070	6630
4	874	2110	3800	4950
5	1110	2030	3800	4590
6	808	2200	4260	6350
7	875	3510	4610	4690
8	901	2510	3820	4970
9	770	1780	3950	5240
10	797	2420	4790	5580
11	1100	2100	4460	5040
12	939	2500	4350	5270
13	664	2860	4210	4930
14	757	3090	4470	5710
15	598	2400	4030	4860
16	558	4700	4280	5530
17	865	2800	4460	4990
18	942	2140	4610	4920
19	1150	6980	3780	6140
20	621	2890	4030	8030
media	847,1	2774	4194	5569,5
diff	-3346,9	-2795,5		

Tabella 1 – Tempi di risposta con e senza la “Capacity Mode” in millisecondi