

POLITECNICO DI TORINO

Dipartimento di ingegneria gestionale e della produzione



Corso di Laurea Triennale in Ingegneria Gestionale

Classe L8 – Ingegneria dell’informazione

SIMULATORE DI TRASPORTO MERCI

Relatore

*Prof. Fulvio Corno
Leo*

Candidato

Emanuele De

247969

Sommario

1. PROPOSTA.....	3
1.1 Descrizione del problema proposto.....	3
1.2 Descrizione della rilevanza gestionale del problema.....	3
1.3 Descrizione dei data-set per la valutazione.....	3
1.4 Descrizione preliminare degli algoritmi coinvolti.....	4
1.5 Descrizione preliminare delle funzionalità previste per l'applicazione software.....	4
3. DIAGRAMMA DELLE CLASSI PRINCIPALI.....	7
4. STRUTTURE DATI.....	9
1) Il package “DAO”.....	9
2) Il package “Model”.....	9
3) Il package “SimulatoreDiTrasportoMerce”.....	14
5. DESCRIZIONE DEL DATA-SET UTILIZZATO.....	14
6. DESCRIZIONE DEGLI ALGORITMI COINVOLTI.....	15
6.1 L'algoritmo di Dijkstra.....	16
6.2 La simulazione.....	18
6.3 L'algoritmo di Yen.....	23
7. ESEMPI DI UTILIZZO DELL'APPLICAZIONE.....	25
8. VALUTAZIONE DEI RISULTATI OTTENUTI E CONCLUSIONI.....	31

1. PROPOSTA

STUDENTE:

Emanuele De Leo

MATRICOLA:

247969

1.1 Descrizione del problema proposto

Il progetto vuole affrontare il problema di gestione dei percorsi per un'azienda di trasporto merci con basi sparse in diverse città italiane attraverso una simulazione delle consegne in una rete di trasporto a grafo. Il trasporto di ogni merce avviene basandosi su un cammino che minimizzi, a seconda dell'esigenza, i costi e/o i tempi impiegati e, allo stesso tempo, che sfrutti in modo accettabile le risorse a disposizione generando una quantità di diversi mezzi di trasporto a seconda delle necessità. Oltre che affrontare un problema di ricerca di cammini minimi ci si vuole concentrare anche sull'analisi dei singoli percorsi, in particolare sulla ricerca di percorsi alternativi di modo da confrontare costi e tempi tra diversi scenari a disposizione.

1.2 Descrizione della rilevanza gestionale del problema

Il problema che si vuole affrontare è un tipico problema di ottimizzazione in una rete di nodi, oltre che essere un problema previsionale. Creando delle flotte di mezzi di trasporto sulla base degli input e degli ordini accolti è possibile avere un utilizzo desiderato delle risorse a disposizione con costi e tempistiche

consoni al tipo di simulazione scelta dall'utente. Infine, scopo dell'applicazione è anche quello di analizzare i singoli percorsi e confrontarli tra di loro, oltre a permettere di confrontare i risultati ottenuti da diverse simulazioni in modo da poter prendere decisioni per il futuro.

1.3 Descrizione dei data-set per la valutazione

Il data-set utilizzato è composto da circa 50000 tratte tra 100 città con relative distanze (riferite alle distanze stradali) e relativi tipi di mezzi di trasporto impiegati; tale data-set viene rielaborato dall'applicazione in base agli input immessi dall'utente.

Il data-set è disponibile al seguente link:

[https://data.world/sapomnia/calcolatore-co2/workspace/file?
filename=Calcolatore+emissioni+CO2.xls](https://data.world/sapomnia/calcolatore-co2/workspace/file?filename=Calcolatore+emissioni+CO2.xls)

1.4 Descrizione preliminare degli algoritmi coinvolti

Gli algoritmi coinvolti sono quelli relativi alla ricerca di cammini minimi tra un nodo sorgente ed uno destinazione all'interno di un grafo tipici della ricerca operativa; in particolare gli algoritmi su cui ci si vuole soffermare sono:

- L'algoritmo di Dijkstra (per la ricerca del cammino minimo in un grafo)
- L'algoritmo di Yen (per la ricerca di k cammini minimi in un grafo)

Il grafo utilizzato viene pesato considerando l'esigenza dell'utente nell'effettuare una simulazione del trasporto che minimizzi il tempo e/o il costo impiegato nelle varie consegne. L'algoritmo di Dijkstra si basa sul peso dato agli archi, tenendo traccia del tipo di mezzo identificato dagli archi stessi.

Una volta completata la simulazione, l'applicazione permette di cercare un percorso alternativo per ogni ordine completato attraverso l'algoritmo di Yen e di visualizzare il percorso migliore (che sarà quello effettivamente intrapreso dall'ordine) ed un

percorso alternativo, permettendo all’utente di confrontare i percorsi in termini di costi, tempi e tratte attraversate.

1.5 Descrizione preliminare delle funzionalità previste per l’applicazione software

1) All’avvio dell’applicazione viene richiesto all’utente di inserire i parametri di input per la creazione del grafo:

- scelta dei mezzi e delle loro caratteristiche (tipo di mezzo, volume e peso massimi trasportabili, velocità media in Km/h e costo allocato in base ai km in €/Km)

- percentuale di peso del costo e del tempo utilizzati per pesare il grafo (una percentuale maggiore del tempo permetterà di effettuare percorsi che minimizzino il tempo impiegato, viceversa nel caso della percentuale di costo); una percentuale 50/50 permette di dare un peso bilanciato tra i due archi del grafo.

2) Cliccando sul bottone “Crea grafo” è possibile generare il grafo in base ai mezzi inseriti ed alle percentuali impostate. Le informazioni sul grafo sono visibili in output.

3) Viene richiesto all’utente di immettere i parametri della simulazione: numero di giorni, numero di ordini giornalieri, lasso di tempo della giornata in cui gli ordini sono accolti, timeout in ore utilizzato per ogni ordine per alzarne la priorità una volta scattato, percentuale di riempimento dello spazio e del peso del mezzo secondo la quale esso può considerarsi pronto per partire o meno.

4) Cliccando sul bottone “Simula” è possibile lanciare la simulazione e vedere il progresso dell’elaborazione su una barra di progresso. In output saranno visibili gli ordini completati in forma tabellare con tutti i dati relativi e, in un’area di testo a parte, l’output generale della simulazione (numero di ordini completati, numero di mezzi creati per ogni tipo e costo complessivo in euro).

5) Inserendo l’ID di un ordine nello spazio sottostante e cliccando sul bottone “Cerca” è possibile visualizzare le informazioni relative a quell’ordine, compreso uno storico che riporta le tappe effettuate

nelle varie città con relativi mezzi coinvolti, date e orari. Scorrendo in basso è possibile osservare il confronto tra il percorso migliore per l'ordine (effettivamente intrapreso) ed il secondo percorso migliore, ognuno con relativi costi e tempi di attraversamento (in minuti) senza considerare il tempo di attesa effettivo in ogni città.

2. Descrizione dettagliata del problema affrontato

Ad oggi molte aziende e privati affidano il trasporto dei propri prodotti ad altre aziende che dispongono di determinati mezzi e che operano in diverse città ricoprendo più aree. L'obiettivo di questa tesi è quello di ricavare informazioni utili ai fini della gestione ottima di un'azienda di trasporto che opera in territorio nazionale attraverso una simulazione delle giornate lavorative; in questo modo è possibile effettuare previsioni su larga scala dell'utilizzo delle risorse a disposizione e dei costi e tempi impiegati.

La simulazione si basa sulla generazione e sul trasporto di diversi ordini (identificati ognuno da un ID, peso, volume, città sorgente, città destinazione ed una data di arrivo) all'interno di un grafo orientato e pesato, il cui peso degli archi (corrispondenti alle tratte presenti nel data-set e identificati dalla distanza, dal tipo di mezzo e dall'ID del mezzo) è calcolato sulla base di una semplice formula matematica che tiene conto dei parametri in input dell'utente. In questo modo viene concesso all'utente di effettuare una simulazione avendo come obiettivo la minimizzazione dei tempi e/o dei costi complessivi.

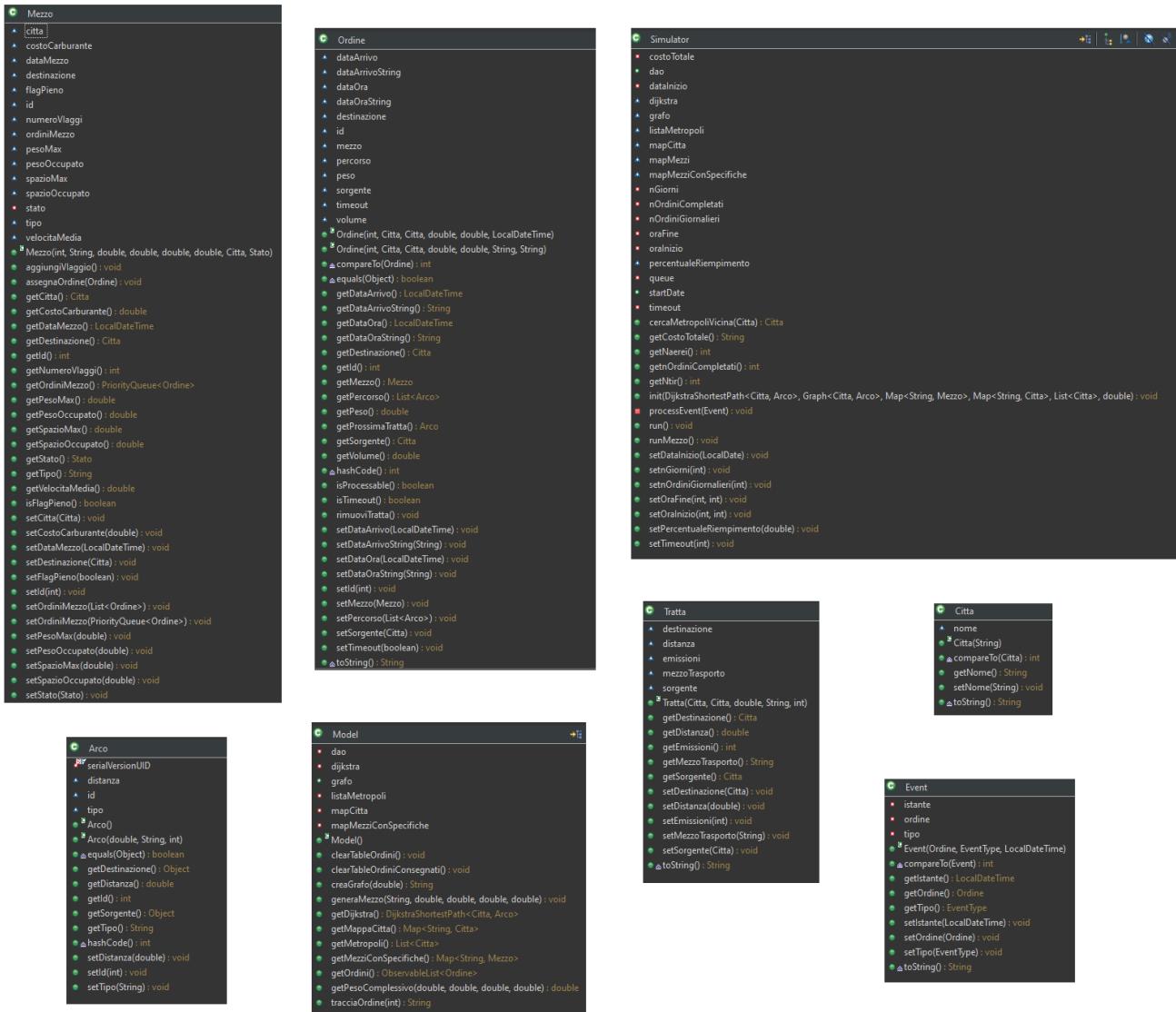
Il peso che viene calcolato per ogni arco è la chiave di volta su cui si basa il trasporto, il quale stabilisce che ogni ordine percorra il tragitto imposto dall'algoritmo di Dijkstra. Ogni volta che arriva un nuovo ordine, il sistema calcola gli archi che esso deve seguire e li applica all'ordine stesso, facendo sì che esso conservi le informazioni sul percorso che dovrà fare.

Un secondo problema di cui ho voluto occuparmi è quello relativo alla ricerca di percorsi alternativi (basati anch'essi sull'algoritmo di Dijkstra) in modo tale da poter effettuare riflessioni sulla differenza tra questi percorsi e i percorsi migliori utilizzati nella simulazione. Per ogni ordine viene mostrato il percorso migliore (effettivamente

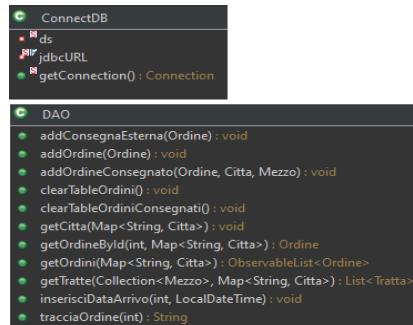
intrapreso) ed il secondo percorso migliore ricavato tramite l'algoritmo di Yen, compreso di tappe, mezzi, peso, costo e tempo impiegato; i risultati ottenuti devono essere in grado di giustificare la scelta degli obiettivi di minimizzazione immessi inizialmente. La simulazione funziona correttamente con il data-set scelto, nonostante esso sia particolarmente vasto e poco pratico. Sarebbe ottimo applicare la simulazione ad un data-set più compatto, con meno tratte e meno città, in modo tale da poterla rendere più fluida nei tempi di processamento e nella gestione degli ordini.

3. Diagramma delle classi principali

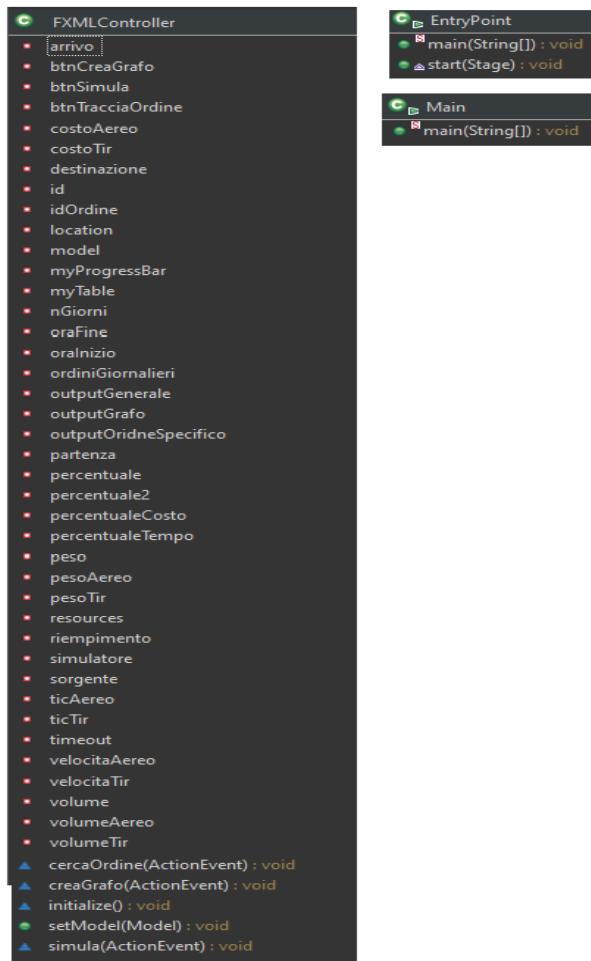
1. Package “**Model**”



2. Package “DAO”



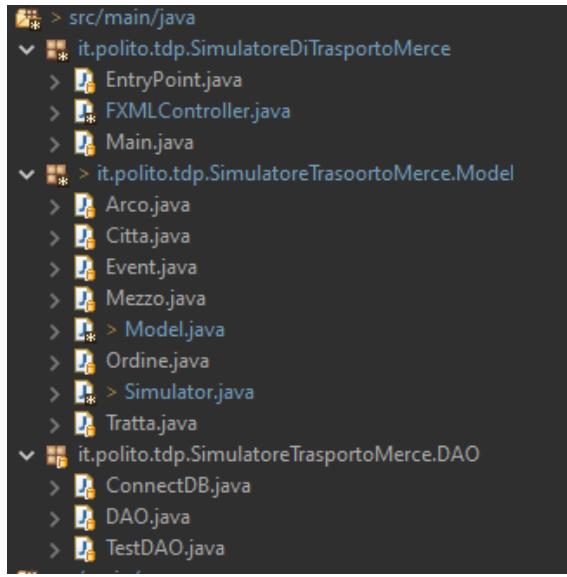
3. Package “SimulatoreDiTrasportoMerce”



4. Strutture dati

L'applicazione è stata realizzata in linguaggio Java e la grafica è stata realizzata in JavaFX grazie all'utilizzo del software SceneBuilder.

Per la logica del codice si è seguito il pattern Model View Controller e il pattern DAO, sono quindi stati sviluppati in 3 pacchetti l'accesso al database (package “DAO”), la logica (pacchetto “Model”) e l’interfaccia utente (pacchetto “SimulatoreDiTrasportoMerce”)



1) Il package “DAO”

- **ConnectDB**: classe con cui viene instaurata la connessione al database locale
- **DAO**: classe che contiene tutti i metodi per la lettura, scrittura e sovrascrittura del database

2) Il package “Model”

CLASSE Model: classe di riferimento dell'applicazione contenente un metodo per creare il grafo, un metodo per pesare ogni arco in base alle percentuali date, uno per generare i mezzi scelti dall'utenti (questi vengono presi come riferimento per la creazione dei mezzi durante la simulazione) e, infine, uno per tracciare un ordine. Il costruttore della classe permette di creare: un oggetto DAO a cui fare riferimento per interagire con il database, una lista di metropoli, una mappa caratteristica dei mezzi (mapMezziConSpecifiche con chiave il tipo di mezzo e valore il mezzo) per tenere traccia delle caratteristiche dei tipi di mezzi che sono creati nella simulazione, una Idmap delle città che viene passata al dao attraverso il metodo getCitta per essere poi restituita carica delle città nel database.

Nel model viene utilizzata una lista “listaMetropoli” a cui sono aggiunte alcune delle più grandi città presenti nel database; questa lista sarà utile nella fase di generazione degli ordini per la simulazione.

```

private Graph<Citta, Arco> grafo;
private DAO dao;
private Map<String, Mezzo> mapMezziConSpecifiche;
private List<Citta> listaMetropoli;
private Map<String, Citta> mapCitta;
private DijkstraShortestPath<Citta, Arco> dijkstra;

public Model() {
    dao = new DAO();
    listaMetropoli = new ArrayList<Citta>(); // METROPOLI : CITTA' IN CUI RISIEDONO GLI AEROPORTI
    mapMezziConSpecifiche = new TreeMap<String, Mezzo>();
    this.mapCitta = new TreeMap<String, Citta>();
    dao.getCitta(mapCitta);
    listaMetropoli.add(mapCitta.get("Roma"));
    listaMetropoli.add(mapCitta.get("Milano"));
    listaMetropoli.add(mapCitta.get("Genova"));
    listaMetropoli.add(mapCitta.get("Torino"));
    listaMetropoli.add(mapCitta.get("Firenze"));
    listaMetropoli.add(mapCitta.get("Bologna"));
    listaMetropoli.add(mapCitta.get("Napoli"));
    listaMetropoli.add(mapCitta.get("Palermo"));

}

```

- **creaGrafo(double percentuale):** permette di creare un grafo orientato e pesato attraverso l’istanza *DirectedWeightedMultigraph<Nodo, Arco>*, dopodichè di aggiungere ad esso le città (*i nodi*) presenti nella mappa delle città (*mapCitta*) attraverso il metodo *Graphs.addAllVertices(grafo, nodi)*. Per aggiungere gli archi vengono estratte dal database le tratte filtrate in base ai tipi di mezzi scelti attraverso il metodo *dao.getTratte(mezzi scelti, mappa città)*. Ogni arco è un’istanza della classe *Arco* che estende la classe *DefaultWeightedEdge* e contiene come informazione aggiuntiva rispetto alle tratte originarie un ID che sarà caratteristico per ogni mezzo (di fatto un arco è un mezzo, in quanto la creazione dei mezzi si baserà sui singoli archi). Se il grafo contiene nodo sorgente e destinazione di una tratta, viene aggiunto un nuovo Arco e ne viene settato il peso attraverso il metodo *getPesoComplessivo(...)*. Una volta che il grafo è completo, viene creata un’istanza della classe *DijkstraShortestPath* e le viene passato il grafo. Il metodo restituisce una stringa con le informazioni del grafo creato.

- **getPesoComplessivo(double distanza, double velocità, double costoCarburante, double percentuale):**
restituisce il peso combinato per ogni arco in base al mezzo, alla distanza e alle percentuali in peso di tempo e costo immesse dall'utente.
Per gestire casi limite ho deciso che con una percentuale di tempo del 15% o meno il peso dell'arco sia rappresentato esclusivamente dal costo in euro, viceversa con una percentuale di costo del 15% o meno il peso sia rappresentato esclusivamente dal tempo di percorrenza.

```
public double getPesoComplessivo(double distanza, double velocita, double costoCarburante, double percentuale) {
    double pesoComplessivo = 0.0;
    if (percentuale <= 15) {
        pesoComplessivo = ((costoCarburante * distanza) * (100 - percentuale));
        return Math.round(pesoComplessivo * 100.00) / 100.00;
    }
    if (percentuale >= 85) {
        pesoComplessivo = ((distanza / velocita)) * percentuale;
        return Math.round(pesoComplessivo * 100.00) / 100.00;
    }
    pesoComplessivo = (((distanza / velocita) * 10) * percentuale)
        + (((costoCarburante * distanza) / 10) * (100 - percentuale)); // PESO
    return Math.round(pesoComplessivo * 100.00) / 100.00;
}
```

NOTA: in realtà costo in euro e tempo di percorrenza hanno unità di misura e ordini di grandezza molto differenti, quindi è naturale aspettarsi che il costo in euro predomini sull'altro dettando esso stesso il peso pur se in percentuale minore. Per ovviare a questo problema ho deciso di ridurre il costo in euro e di aumentare il valore del tempo in modo da tentare di portare entrambi su una scala più equa possibile.

- **generaMezzo(String tipo, double pesoMax, double spazioMax, double velocitàMedia, double costoCarburante):** permette di generare un tipo di mezzo che fungerà da modello per la creazione dei mezzi dello stesso tipo durante la simulazione; il mezzo generato entrerà nella mappa “mapMezziConSpecifiche” (eventualmente sostituendo un mezzo dello stesso tipo precedentemente aggiunto).
- **TracciaOrdine(int id):** restituisce una stringa con le informazioni sull'ordine (il cui ID viene passato) ed uno storico del suo percorso letto dal database nella tabella “ordini_consegnati” attraverso il metodo dao.tracciaOrdine(id).

*Il metodo permette inoltre di confrontare il percorso migliore intrapreso dall'ordine con il secondo miglior percorso trovato grazie all'istanza YenKShortestPath<Nodo,Arco> a cui viene passato come parametro il grafo e che rilascia i percorsi come lista di GraphPath attraverso il metodo
 getPaths(nodoSorgente, nodoDestinazione, k=2); per il confronto tra i percorsi il metodo calcola per ognuno di essi i pesi complessivi, il costo totale (in euro), il tempo totale (in minuti) permettendo di apprezzarne le diversità.*

```

public String tracciaordine(int id) {
    String output = "";
    Ordine ordineTracciato = dao.getOrdineById(id, mapCitta);
    output += ordineTracciato + "\n\n";
    output += "STORICO\n";
    output += dao.tracciaOrdine(id) + "\n\n";

    KShortestPathAlgorithm<Città, Arco> pathInspector = new YenKShortestPath<Città, Arco>(grafo); // IMPLEMENTAZIONE
                                                                 // ALGORITMO DI
                                                                 // YEN
    List<GraphPath<Città, Arco>> paths = pathInspector.getPaths(ordineTracciato.getsorgente(),
        ordineTracciato.getdestinazione(), 2); // LISTA DI K SHORTEST PATHS

    List<Arco> edgeBestPath = paths.get(0).getEdgeList();
    List<Arco> edgeBestSecondPath = paths.get(1).getEdgeList();

    output += "PERCORSO MIGLIORE: \n";
    // System.out.println("Best path" + edgeBestPath);

    double costo1 = 0.0;
    double costoEuro1 = 0.0;
    LocalDateTime dataArrivo = ordineTracciato.getDataOra();

    for (Arco arco : edgeBestPath) {
        output += "" + (Città) arco.getsorgente() + " - " + (Città) arco.getdestinazione() + " mezzo="
            + arco.getTipo().replace("Autobus", "Tir") + " id=" + arco.getId() + "\n";
        costo1 += getPesoComplessivo(arco.getDistanza(),
            mapMezziConSpecifiche.get(arco.getTipo()).getVelocitaMedia(),
            mapMezziConSpecifiche.get(arco.getTipo()).getCostoCarburante(), 50);
        costoEuro1 += arco.getDistanza() * mapMezziConSpecifiche.get(arco.getTipo()).getCostoCarburante();
        ordineTracciato.setDataOra(ordineTracciato.getDataOra().plusSeconds(Math.round(
            (arco.getDistanza()) / mapMezziConSpecifiche.get(arco.getTipo()).getVelocitaMedia()) * 3600)));
    }

    output += "\nPESO: " + Math.round(costo1 * 100.0) / 100.0 + " COSTO: "
        + Math.round(costoEuro1 * 100.00) / 100.00 + " € DURATA: "
        + Duration.between(dataArrivo, ordineTracciato.getDataOra().toMinutes()) + " minuti\n\n";

    output += "PERCORSO ALTERNATIVO: \n";
    double costo2 = 0.0;
    double costoEuro2 = 0.0;

    Ordine ordineTracciato2 = dao.getOrdineById(id, mapCitta);

    for (Arco arco : edgeBestSecondPath) {
        output += "" + (Città) arco.getsorgente() + " - " + (Città) arco.getdestinazione() + " mezzo="
            + arco.getTipo().replace("Autobus", "Tir") + " id=" + arco.getId() + "\n";
        costo2 += getPesoComplessivo(arco.getDistanza(),
            mapMezziConSpecifiche.get(arco.getTipo()).getVelocitaMedia(),
            mapMezziConSpecifiche.get(arco.getTipo()).getCostoCarburante(), 50);
        costoEuro2 += arco.getDistanza() * mapMezziConSpecifiche.get(arco.getTipo()).getCostoCarburante();
        ordineTracciato2.setDataOra(ordineTracciato2.getDataOra().plusSeconds(Math.round(
            (arco.getDistanza()) / mapMezziConSpecifiche.get(arco.getTipo()).getVelocitaMedia()) * 3600)));
    }

    output += "\nPESO: " + Math.round(costo2 * 100.0) / 100.0 + " COSTO: "
        + Math.round(costoEuro2 * 100.00) / 100.00 + " € DURATA: "
        + Duration.between(dataArrivo, ordineTracciato2.getDataOra().toMinutes()) + " minuti";

    output.replace("Autobus", "Tir");
    return output;
}

```

- **getMezziConSpecifiche():** ritorna la mappa di modello dei mezzi (`mapMezziConSpecifiche`).
- **getOrdini():** legge gli ordini dal database passando dal dao e li restituisce in una `ObservableList<Ordine>` in modo da poter essere utilizzati nella tabel view del controller.
- **getDijkstra():** restituisce l'istanza del `DijkstraShortestPath` applicata al grafo
- **clearTableOrdini() , clearTableOrdiniConsegnati() :** svuotano le suddette tabelle nel database
- **getMetropoli():** restituisce le metropoli (grandi città in cui ci sono gli aeroporti)
- **getMappaCittà():** restituisce la mappa delle citt

CLASSE Simulator: classe che implementa la simulazione costituita dai metodi per la sua inizializzazione ed esecuzione

```

public class Simulator {

    public Date startDate = null;
    public DAO dao;
    Graph<Citta, Arco> grafo;
    DijkstrashortestPath<Citta, Arco> dijkstra;
    Map<String, Mezzo> mapMezziConSpecifiche;
    Map<Integer, Mezzo> mapMezzi;
    Map<String, Citta> mapCitta;
    double percentualeRiempimento;
    List<Citta> listaMetropoli;

    // Eventi

    private PriorityQueue<Event> queue;

    // Parametri

    private int nOrdiniGiornalieri;
    private int nGiorni;
    private LocalDate dataInizio;
    private LocalTime oraInizio;
    private LocalTime oraFine;
    private int timeout;

    // Stato del mondo

    private int nOrdiniCompletati;
    private double costoTotale;
}

```

- ***init()***: imposta i parametri iniziali immessi dall'utente e simula ordini da aggiungere alla coda degli eventi. Pesi e volumi degli ordini sono randomizzati prendendo in considerazione le capacità dei mezzi, mentre le sorgenti e le destinazioni sono generate con una probabilità diversa a seconda delle città: la probabilità che una città sorgente non sia una metropoli è di 1/3, mentre la probabilità che una città destinazione non sia una metropoli è impostata a 1/4. Si sono scelti questi valori di probabilità per rendere più verosimili possibili gli output, in quanto è nelle metropoli che vi deve essere più traffico di ordini.
- ***run()***: esegue la simulazione processando gli eventi nella coda degli eventi e, ogni volta che passano 5 millisecondi, lancia il metodo *runMezzo()*
- ***runMezzo()***: esegue il controllo dei mezzi e, se alcuni di questi possono partire, li fa partire aggiornando le informazioni sugli ordini che trasportano e svuotandosi

(la logica della simulazione viene trattata più nel dettaglio nella parte dedicata agli algoritmi).

3) Il package “**SimulatoreDiTrasportoMerce**”

Questo è il package che contiene il controllo del sistema tramite l'interfaccia utente.

Esso è composto dalla classe principale Main.java per lanciare il software, la classe EntryPoint.java che permette di applicare lo scenario grafico e, infine, la classe FXMLController.java utilizzata per il controllo da parte dell'utente e funge da collegamento tra la logica applicativa e l'interfaccia grafica.

5. Descrizione del data-set utilizzato

Il data-set utilizzato (reperibile al link:

[https://data.world/sapomnia/calcolatore-co2/workspace/file?
filename=Calcolatore+emissioni+CO2.xls](https://data.world/sapomnia/calcolatore-co2/workspace/file?filename=Calcolatore+emissioni+CO2.xls)

) è basato sulle emissioni di diversi tipi di mezzi di trasporto per diverse tratte stradali tra delle città italiane; esso è composto da una tabella “tratte” contenente diversi percorsi tra città con relativa distanza (in Km), relativo tipo di mezzo con cui quel percorso è effettuato ed il valore di emissioni complessivo (in grammi di CO2).

#	Nome	Tipo di dati
1	Partenza	VARCHAR
2	Destinazione	VARCHAR
3	Distanza_km	VARCHAR
4	Mezzo_di_trasporto	VARCHAR
5	Emissioni_g	INT

Nella struttura del data-set utilizzato dal software sono presenti, inoltre, 2 tabelle vuote utilizzate durante l'esecuzione della simulazione e la visualizzazione dei dati in output.

- TABELLA “ordini”:

contiene i dati degli ordini simulati in tempo reale durante la simulazione. Questa tabella è utilizzata per leggere e sovrascrivere lo stato degli ordini mantenendo le informazioni caratteristiche degli ordini stessi. Per ogni ordine sono presenti i campi: ID, peso (Kg),

volume(m^3), città sorgente, città destinazione, data di partenza (data in cui viene accolto l'ordine dal sistema), data di arrivo (NULL se l'ordine non è ancora arrivato a destinazione, altrimenti la data di consegna)

#	Nome	Tipo di dati
1	ID	INT
2	Peso	DOUBLE
3	Volume	DOUBLE
4	Sorgente	VARCHAR
5	Destinazione	VARCHAR
6	dataPartenza	TIMESTAMP
7	dataArrivo	TIMESTAMP

- TABELLA “ordini_consegnati”:

E’ utilizzata per registrare gli eventi di consegne intermedie per ogni ordine prima di giungere a destinazione. Durante la simulazione questa tabella viene riempita con righe contenenti l’ID dell’ordine, la città di partenza (o di consegna) e relativi data e mezzo coinvolto. Questa tabella sarà utile alla fine della simulazione quando sarà possibile tracciare i passi che ha fatto un ordine tramite il suo ID.

#	Nome	Tipo di dati
1	ID	INT
2	citta_consegna	VARCHAR
3	data	TIMESTAMP
4	ID_mezzo	INT
5	tipo_mezzo	VARCHAR

6. Descrizione degli algoritmi coinvolti

Oltre che sull'algoritmo legato alla logica della simulazione, l'applicazione si basa su algoritmi per la ricerca di cammini minimi in una rete di nodi. Questi sono: l'algoritmo di Dijkstra per la ricerca di un cammino minimo tra un nodo sorgente e uno destinazione, l'algoritmo di Yen per la ricerca di k cammini minimi tra un nodo sorgente e uno destinazione.

6.1 L'ALGORITMO DI DIJKSTRA

L'algoritmo di Dijkstra è utilizzato per la risoluzione del problema del cammino minimo su un grafo pesato. Il problema del cammino minimo è legato alla ricerca di un percorso tra un nodo sorgente ed uno destinazione tale che il peso complessivo degli archi che lo compongono sia minimo.

I passi dell'algoritmo di Dijkstra

1) Inizializzazione:

- i : numero di nodi
- $S(i-1)$: insieme dei nodi visitabili del grafo (escluso il nodo sorgente)
- $P(i)$ = pesi degli archi che collegano il nodo sorgente a tutti i nodi del grafo (se il nodo sorgente non è collegato al nodo in questione attraverso un percorso, il peso di esso viene posto uguale a infinito, mentre il peso dal nodo 0 al nodo 0 è posto a 0)
- $Path(i-1)$: percorso di nodi minimo trovato

2) Esecuzione:

Trova un nodo j visitabile presente in S tale che il peso dell'arco che lo congiunge al nodo sorgente sia minimo, lo rimuove da S e lo aggiunge al Path.

3) Iterazione:

- trova i nodi in S raggiungibili dall'ultimo nodo visitato e calcola i pesi complessivi tra questi ed il nodo sorgente
- aggiorna P con i nuovi pesi minimi corrispondenti ai cammini verso ogni nodo raggiungibile dal nodo sorgente
- sceglie il nodo j raggiungibile presente in S il cui peso complessivo degli archi che lo collegano al nodo sorgente sia il minimo in assoluto
- rimuove il nodo j trovato da S .

4) Terminazione

L'algoritmo termina quando l'ultimo nodo visitato non può raggiungere nodi in S ed S non ha più nodi visitabili.

Nell'applicazione ho deciso di implementare l'algoritmo di Dijkstra utilizzando la classe DijkstraShortestPath che implementa l'interfaccia ShortestPathAlgorithm messa a disposizione dalla libreria JGraphT; essa permette di applicare al grafo l'algoritmo in modo da trovare comodamente il cammino minimo per il trasporto di un ordine.

La classe DijkstraShortestPath è dichiarata nella classe Model e viene creata un'istanza di essa al momento di creazione del grafo; la classe prende come parametro il grafo stesso, dopodiché restituisce il cammino minimo (nel caso dell'applicazione una lista di archi) o peso complessivo del cammino grazie ai metodi messi a disposizione.

```

public String creaGrafo(double percentuale) {
    int codiciMezzi = 1;
    Arco arco = null;
    this.grafo = new DirectedWeightedMultigraph<Citta, Arco>(Arco.class);
    Graphs.addAllVertices(this.grafo, mapCitta.values());

    for (Tratta t : dao.getTratte(mapMezziConSpecifiche.values(), mapCitta)) {
        if (grafo.containsVertex(t.getSorgente()) && grafo.containsVertex(t.getDestinazione())) {
            arco = grafo.addEdge(t.getSorgente(), t.getDestinazione());
            arco.setDistanza(t.getDistanza());
            arco.setId(codiciMezzi);
            arco.setTipo(t.getMezzoTrasporto());

            grafo.setEdgeWeight(arco,
                this.getPesoComplessivo(t.getDistanza(),
                    mapMezziConSpecifiche.get(arco.getTipo()).getVelocitaMedia(),
                    mapMezziConSpecifiche.get(arco.getTipo()).getCostoCarburante(), percentuale));
            codiciMezzi++;
        }
    }
    dijkstra = new DijkstraShortestPath<Citta, Arco>(grafo);
    return String.format(" GRAFO CREATO\n\n - %d vertici\n - %d archi\n", this.grafo.vertexSet().size(),
        this.grafo.edgeSet().size());
}

```

Poiché il grafo è pesato tenendo conto delle percentuali di peso che l'utente vuole applicare (al costo ed al tempo impiegato), in questo caso l'algoritmo permette non solo di trovare il cammino minimo, ma anche di trovare il cammino con il costo minimo (in termini di costo in € o in termini di tempo). La funzione *getPesoComplessivo()* ritorna il peso dato ad ogni arco considerando la distanza della tratta, la velocità del mezzo in questione, il costo del mezzo ed una percentuale che l'utente immette in input riferita al peso percentuale che egli intende dare al tempo nel calcolo (viceversa 100-percentuale sarà il peso percentuale del costo).

L'istanza *DijkstraShortestPath* viene passata alla classe *Simulator* e viene utilizzata durante la simulazione per la ricerca dei cammini per ogni ordine accolto.

Nello screen sottostante si può notare la gestione di un ordine (*currentOnDB*) letto dal database e in fase di elaborazione.

```

case NUOVO_ORDINE:
    currentOnDB.setPercorso(dijkstra.getPath(currentOnDB.getSorgente(), currentOnDB.getDestinazione()).getEdgeList());

```

6.2 LA SIMULAZIONE

La simulazione è implementata dalla classe Simulator, che contiene delle variabili passate dalla classe Model durante l'inizializzazione (come il grafo, il DijkstraShortestPath, i tipi di mezzi scelti, le città) e delle variabili legate alla simulazione stessa (come la coda degli eventi, i parametri in input, variabili di stato del sistema, una data di inizio della simulazione “startDate”).

Il metodo init()

Il metodo imposta i parametri iniziali, crea un oggetto Date “startDate” corrispondente all'istante di inizio, dopodiché simula gli ordini, li aggiunge sul database nella tabella “ordini” generando eventi del tipo NUOVO_ORDINE ed aggiungendoli alla coda degli eventi. Gli eventi contengono l'ordine, il tipo di evento e l'istante in cui avvengono.

Il metodo run()

Il metodo permette di avviare la simulazione. Una volta lanciato viene creato un flag=true utile a capire quando la coda degli eventi è vuota e non ci sono più eventi da processare; finché flag=true e la coda degli eventi non è vuota vengono estratti gli eventi e processati grazie al metodo *processEvent()*.

Il trasporto effettivo degli ordini avviene ogni volta che l'istante currentDate creato all'inizio del ciclo while differisce dall'istante startDate generato nel metodo init() di 5 o più millisecondi; a questo punto viene lanciato il controllo dei mezzi e quelli che soddisfano i vincoli di partenza possono partire.

```

public void run() {
    boolean check = true;
    while (check) {
        Date currentDate = new Date();
        long diffInMillies = Math.abs(currentDate.getTime() - startDate.getTime());
        long diff = TimeUnit.SECONDS.convert(diffInMillies, TimeUnit.MILLISECONDS);
        if (diff >= 5) {
            runMezzo();
        }
        if (!this.queue.isEmpty()) {
            Event nuovoEvento = this.queue.poll();
            processEvent(nuovoEvento);
        } else {
            check = false;
        }
    }
}

```

Il metodo *processEvent()*

Il metodo processa gli eventi estratti dalla coda degli eventi e li gestisce a seconda del tipo di evento.

```

Ordine nuovoOrdine = e.getOrdine();
Ordine currentOnDB = dao.getOrdineById(nuovoOrdine.getId(), mapCitta);

switch (e.getTipo()) {
    case NUOVO_ORDINE:
        currentOnDB.setPercorso(dijkstra.getPath(currentOnDB.getSorgente(), currentOnDB.getDestinazione()).getEdgeList());
        Arco prossimo = currentOnDB.getProssimaTratta();

        if (!mapMezzi.containsKey(prossimo.getId())) {
            mapMezzi.put(prossimo.getId(),
                new Mezzo(prossimo.getId(), prossimo.getTipo(),
                    mapMezziconSpecifiche.get(prossimo.getTipo()).getPesoMax(),
                    mapMezziconSpecifiche.get(prossimo.getTipo()).getSpazioMax(),
                    mapMezziconSpecifiche.get(prossimo.getTipo()).getVelocitaMedia(),
                    mapMezziconSpecifiche.get(prossimo.getTipo()).getCostoCarburante(),
                    (Citta) prossimo.getSorgente(), Stato.DISPONIBILE));
            mapMezzi.get(prossimo.getId()).setDestinazione((Citta) prossimo.getDestinazione());
        }

        currentOnDB.setMezzo(mapMezzi.get(prossimo.getId()));

        if (currentOnDB.isProcessabile()) {
            currentOnDB.getMezzo().assegnaOrdine(currentOnDB);
        } else {
            this.queue.add(
                new Event(currentOnDB, EventType.ORDINE_IN CORSO, currentOnDB.getDataora().plusHours(timeout)));
        }
    break;
}

```

Se l'evento è del tipo NUOVO_ORDINE viene settato il percorso che dovrà fare attraverso il DijkstraShortestPath, dopodiché viene estratta la prima tratta che l'ordine dovrà percorrere. La tratta estratta corrisponde ad un arco che ha come informazione l'ID e il tipo di mezzo incaricato di prendere l'ordine in questione. Se la mappa dei mezzi non contiene l'ID, viene creato e aggiunto un mezzo alla mappa con chiave l'ID dell'arco e con le caratteristiche

del tipo di mezzo creato, dopodiché gli viene inserita la destinazione della tratta di cui si occuperà da qui in avanti. A questo punto viene settato il mezzo all'ordine in questione (in modo tale che l'ordine mantenga l'informazione del mezzo che deve prenderlo); sta poi all'ordine controllare che il mezzo sia disponibile attraverso il metodo `isProcessable()` della classe `Ordine`.

```
public boolean isProcessable() {  
  
    if (mezzo.getStato().equals(Mezzo.Stato.DISPONIBILE)  
        && mezzo.getPesoOccupato() + this.peso <= mezzo.getPesoMax()  
        && mezzo.getSpazioOccupato() + this.volume <= mezzo.getSpazioMax()) {  
        return true;  
    }  
  
    if (mezzo.getStato().equals(Mezzo.Stato.DISPONIBILE)) {  
        mezzo.setFlagPieno(true);  
    }  
    return false;  
}
```

Quando il mezzo è disponibile e non è pieno l'ordine viene assegnato al mezzo in questione attraverso il metodo `assegnaOrdine()` della classe `Mezzo`, altrimenti viene generato l'evento `ORDINE_IN_CORSO` che scatterà dopo il tempo di timeout e permetterà all'ordine di riprovare ad essere preso dal mezzo incaricato di prenderlo.

Quando arriva un evento `ORDINE_IN_CORSO` il metodo `processEvent()` controlla la sua prossima tratta. Se questa è nulla vuol dire che l'ordine è stato trasportato ed è arrivato a destinazione prima che scattasse il timeout, perciò esso viene considerato consegnato. Se la prossima tratta non è nulla vengono aggiunte le ore di timeout al mezzo e viene posto il flag di timeout interno all'ordine uguale a TRUE.

Una volta finiti i controlli il metodo procede cercando di piazzare l'ordine su un mezzo come nel caso dell'evento `NUOVO_ORDINE` e, se non riesce a piazzarlo, genera nuovamente un evento `ORDINE_IN_CORSO` per riprovare dopo un ulteriore timeout.

```

case ORDINE_IN_CORSO:

    Arco passo = nuovoOrdine.getProssimaTratta();

    if (passo == null) {
        dao.addOrdineConsegnato(nuovoOrdine, nuovoOrdine.getDestinazione(), nuovoOrdine.getMezzo());
        dao.inserisciDataArrivo(nuovoOrdine.getId(), nuovoOrdine.getDataOra());
        break;
    }

    nuovoOrdine.setDataOra(nuovoOrdine.getDataOra().plusHours(timeout));
    nuovoOrdine.setTimeout(true);

    if (!mapMezzi.containsKey(passo.getId())) {
        mapMezzi.put(passo.getId(),
            new Mezzo(passo.getId(), passo.getTipo(),
                mapMezziConSpecifiche.get(passo.getTipo()).getPesoMax(),
                mapMezziConSpecifiche.get(passo.getTipo()).getSpazioMax(),
                mapMezziConSpecifiche.get(passo.getTipo()).getVelocitaMedia(),
                mapMezziConSpecifiche.get(passo.getTipo()).getCostoCarburante(),
                (Città) passo.getSorgente(), Stato.DISPONIBILE));
        mapMezzi.get(passo.getId()).setDestinazione((Città) passo.getDestinazione());
    }

    nuovoOrdine.setMezzo(mapMezzi.get(passo.getId()));

    if (!mapMezzi.get(passo.getId()).getOrdiniMezzo().contains(nuovoOrdine)) {
        if (nuovoOrdine.isProcessabile()) {

            mapMezzi.get(passo.getId()).assegnaOrdine(nuovoOrdine);
            mapMezzi.get(passo.getId()).setDataMezzo(nuovoOrdine.getDataOra());

            } else {
                this.queue.add(new Event(nuovoOrdine, EventType.ORDINE_IN_CORSO,
                    nuovoOrdine.getDataOra().plusHours(timeout)));
            }
        }
        break;
    }
}

```

Il metodo runMezzo()

Quando la startDate creata nel metodo init() differisce di almeno 5 millisecondi dalla data corrente creata all'inizio del ciclo while nel metodo run() viene lanciato il metodo *runMezzo()*. Questo metodo si occupa di effettuare un ciclo sulla mappa dei mezzi tramite gli ID in modo tale da controllare lo stato di ogni mezzo creato precedentemente. Così' se un mezzo viene trovato occupato, viene cambiato il suo stato a disponibile; se il mezzo è trovato disponibile e rispetta i vincoli per partire (ovvero è pieno o rispetta le condizioni di riempimento inserite dall'utente) può partire e verrà considerato occupato.

Quando un mezzo parte vengono estratti uno alla volta gli ordini presenti nella sua coda degli ordini e vengono aggiornati con le informazioni del viaggio compiuto:

- Il flag del timeout viene posto uguale a FALSE;
- La data dell'ordine viene sostituita da quella di partenza del mezzo e vengono caricate le informazioni sulla sua partenza sul database nella tabella ordini_consegnati;
- Viene calcolato il tempo di viaggio e viene aggiunto alla data dell'ordine, dopodichè vengono nuovamente caricate sul database le informazioni relative all'arrivo

Se la città destinazione del mezzo corrisponde alla città destinazione dell'ordine, l'ordine ha completato il suo percorso e viene aggiornato il suo campo nella tabella ordini con la data di arrivo.

Se la città destinazione del mezzo non corrisponde alla città destinazione dell'ordine vuol dire che esso ha compiuto uno step e deve passare al successivo, quindi viene aggiornata la sua sorgente con la città in cui si trova, viene rimossa la prima tratta presente nella lista del suo percorso, viene caricata sul database l'informazione sulla tappa e, infine, viene generato un nuovo evento del tipo ORDINE_IN_CORSO in modo tale che esso continui il percorso fino alla sua destinazione.

```

public void runMezzo() {
    long time = 0;
    double distanza = 0.0;

    for (int id : mapMezzi.keySet()) {
        if (mapMezzi.get(id).getStato().equals(Stato.OCCUPATO)) {
            mapMezzi.get(id).setStato(Stato.DISPONIBILE);
        }

        if ((mapMezzi.get(id).getStato().equals(Stato.DISPONIBILE)
                && mapMezzi.get(id).getSpazioOccupato() >= (percentualeRiempimento / 100)
                * mapMezzi.get(id).getSpazioMax())
                && mapMezzi.get(id).getPesoOccupato() >= (percentualeRiempimento / 100)
                * mapMezzi.get(id).getPesoMax())
                || mapMezzi.get(id).isFlagPieno() == true) {

            mapMezzi.get(id).setStato(Stato.OCCUPATO);
            mapMezzi.get(id).setFlagPieno(false);
            mapMezzi.get(id).aggiungiViaggio();
            costoTotale += mapMezzi.get(id).getOrdiniMezzo().peek().getProssimaTratta().getDistanza()
                * mapMezzi.get(id).getCostoCarburante();

            while (!mapMezzi.get(id).getOrdiniMezzo().isEmpty()) {
                Ordine ordineDaGestire = mapMezzi.get(id).getOrdiniMezzo().poll();
                ordineDaGestire.setTimeout(false);
                ordineDaGestire.setDataOra(mapMezzi.get(id).getDataMezzo());
                dao.addOrdineConsegnato(ordineDaGestire, mapMezzi.get(id).getCitta(), mapMezzi.get(id));
                long secondi = Math.round(
                    (ordineDaGestire.getProssimaTratta().getDistanza() / mapMezzi.get(id).getVelocitaMedia())
                    * 3600);
                distanza = (double) ordineDaGestire.getProssimaTratta().getDistanza();
                time = secondi;
                ordineDaGestire.setDataOra(mapMezzi.get(id).getDataMezzo().plusSeconds(secondi));

                if (mapMezzi.get(id).getDestinazione().equals(ordineDaGestire.getDestinazione())) {
                    System.out.println("ordine " + ordineDaGestire.getId() + " CONSEGNATO");
                    dao.addOrdineConsegnato(ordineDaGestire, ordineDaGestire.getDestinazione(), mapMezzi.get(id));
                    dao.inserisciDataArrivo(ordineDaGestire.getId(), ordineDaGestire.getDataOra());
                    ordiniInCompletati++;
                } else {
                    System.out.println(
                        "step ordine= " + ordineDaGestire.getId() + " a " + mapMezzi.get(id).getDestinazione());
                    ordineDaGestire.setSorgente(mapMezzi.get(id).getDestinazione());
                    dao.addOrdineConsegnato(ordineDaGestire, mapMezzi.get(id).getDestinazione(), mapMezzi.get(id));
                    ordineDaGestire.rimuoviTratta();
                    this.queue.add(new Event(ordineDaGestire, EventType.ORDINE_IN_CORSO,
                        ordineDaGestire.getDataOra().plusHours(timeout)));
                }
            }
            costoTotale += distanza * mapMezzi.get(id).getCostoCarburante();
            mapMezzi.get(id).setDataMezzo(mapMezzi.get(id).getDataMezzo().plusSeconds(time * 2));
            mapMezzi.get(id).setPesoOccupato(0.0);
            mapMezzi.get(id).setSpazioOccupato(0.0);
        }
    }
}

```

Alla fine dell'estrazione degli ordini, viene aggiunto il costo della tratta al costo totale, viene aggiornata la data del mezzo (questa deve tenere conto del fatto che il mezzo torna nella sua città di partenza dopo la consegna) e vengono azzerati i suoi valori di volume e peso occupato in quanto esso si è svuotato.

6.3 L'ALGORITMO DI YEN

L'algoritmo di Yen è utilizzato per la risoluzione del problema della ricerca di k cammini minimi tra un nodo sorgente ed uno destinazione in un grafo pesato.

I passi dell'algoritmo di Yen

1) Inizializzazione:

- 6.** Paths[i]: lista dei cammini minimi migliori in ordine di costo crescente
- 7.** B[]: lista dei potenziali cammini minimi per la ricerca dei migliori

Il primo passo dell'algoritmo si basa sull'applicazione dell'algoritmo di Dijkstra per trovare il cammino minimo migliore tra il nodo sorgente e il nodo destinazione. Il cammino trovato entra a fare parte della lista dei migliori cammini

2) Iterazione:

Trovato il primo cammino, l'algoritmo prevede che vengano trovati dei potenziali cammini minimi alternativi da aggiungere alla lista B via via che si escludono gli archi dei migliori cammini minimi trovati in ordine partendo dal nodo radice che si prende in considerazione. Tra i cammini trovati, quello che risulta avere costo minimo entra a far parte della lista dei k cammini minimi migliori.

3) Terminazione:

L'algoritmo termina quando non è più possibile trovare potenziali cammini minimi da inserire nella lista B, quindi quando la lista B risulta vuota.

L'algoritmo è implementato nel metodo **tracciaOrdine()** della classe Model attraverso la creazione di un'istanza della classe YenKShortestPath, la quale implementa a sua volta l'interfaccia KShortestPathAlghoritm messa a disposizione dalla libreria JGraphT. Per applicare l'algoritmo è necessario passare come parametro il grafo al YenKShortestPath, dopodiché la classe ritornerà i k cammini minimi (in questo caso k=2 in quanto si cercano i primi due migliori cammini) tra un nodo sorgente ed uno destinazione sotto forma di una lista di oggetti GraphPath. I metodi offerti dalla classe KShortestPathAlghoritm permettono di ricavare i percorsi in liste di archi o di nodi del grafo.

```

public String tracciaOrdine(int id) {
    String output = "";
    Ordine ordineTracciato = dao.getOrdineById(id, mapCitta);
    output += "STORICO\n";
    output += dao.tracciaOrdine(id) + "\n\n";
    KShortestPathAlgorithm<Città, Arco> pathInspector = new YenKShortestPath<Città, Arco>(grafo); // IMPLEMENTAZIONE
                                                                 // ALGORITMO DI
                                                                 // YEN
    List<GraphPath<Città, Arco>> paths = pathInspector.getPaths(ordineTracciato.getSorgente(),
        ordineTracciato.getDestinazione(), 2); // LISTA DI K SHORTEST PATHS
    List<Arco> edgeBestPath = paths.get(0).getEdgeList();
    List<Arco> edgeBestSecondPath = paths.get(1).getEdgeList();

    output += "PERCORSO MIGLIORE: \n";
    // System.out.println("Best path" + edgeBestPath);

    double costo1 = 0.0;
    double costoEuro1 = 0.0;
    LocalDateTime dataArrivo = ordineTracciato.getDataOra();

    for (Arco arco : edgeBestPath) {
        output += "" + (Città) arco.getsorgente() + " - " + (Città) arco.getDestinazione() + " mezzo="
            + arco.getTipo().replace("Autobus", "Tir") + " id=" + arco.getId() + "\n";
        costo1 += getPesoComplessivo(arco.getDistanza()),
            mapMezziConSpecifiche.get(arco.getTipo()).getVelocitaMedia(),
            mapMezziConSpecifiche.get(arco.getTipo()).getCostoCarburante(), 50);
        costoEuro1 += arco.getDistanza() * mapMezziConSpecifiche.get(arco.getTipo()).getCostoCarburante();
        ordineTracciato.setDataOra(ordineTracciato.getDataOra().plusSeconds(Math.round(
            (arco.getDistanza() / mapMezziConSpecifiche.get(arco.getTipo()).getVelocitaMedia()) * 3600)));
    }

    output += "\nPESO: " + Math.round(costo1 * 100.0) / 100.0 + " COSTO: "
        + Math.round(costoEuro1 * 100.00) / 100.00 + " € DURATA: "
        + Duration.between(dataArrivo, ordineTracciato.getDataOra()).toMinutes() + " minuti\n\n";

    output += "PERCORSO ALTERNATIVO: \n";
    double costo2 = 0.0;
    double costoEuro2 = 0.0;

    Ordine ordineTracciato2 = dao.getOrdineById(id, mapCitta);

    for (Arco arco : edgeBestSecondPath) {
        output += "" + (Città) arco.getsorgente() + " - " + (Città) arco.getDestinazione() + " mezzo="
            + arco.getTipo().replace("Autobus", "Tir") + " id=" + arco.getId() + "\n";
        costo2 += getPesoComplessivo(arco.getDistanza()),
            mapMezziConSpecifiche.get(arco.getTipo()).getVelocitaMedia(),
            mapMezziConSpecifiche.get(arco.getTipo()).getCostoCarburante(), 50);
        costoEuro2 += arco.getDistanza() * mapMezziConSpecifiche.get(arco.getTipo()).getCostoCarburante();
        ordineTracciato2.setDataOra(ordineTracciato2.getDataOra().plusSeconds(Math.round(
            (arco.getDistanza() / mapMezziConSpecifiche.get(arco.getTipo()).getVelocitaMedia()) * 3600)));
    }

    output += "\nPESO: " + Math.round(costo2 * 100.0) / 100.0 + " COSTO: "
        + Math.round(costoEuro2 * 100.00) / 100.00 + " € DURATA: "
        + Duration.between(dataArrivo, ordineTracciato2.getDataOra()).toMinutes() + " minuti";

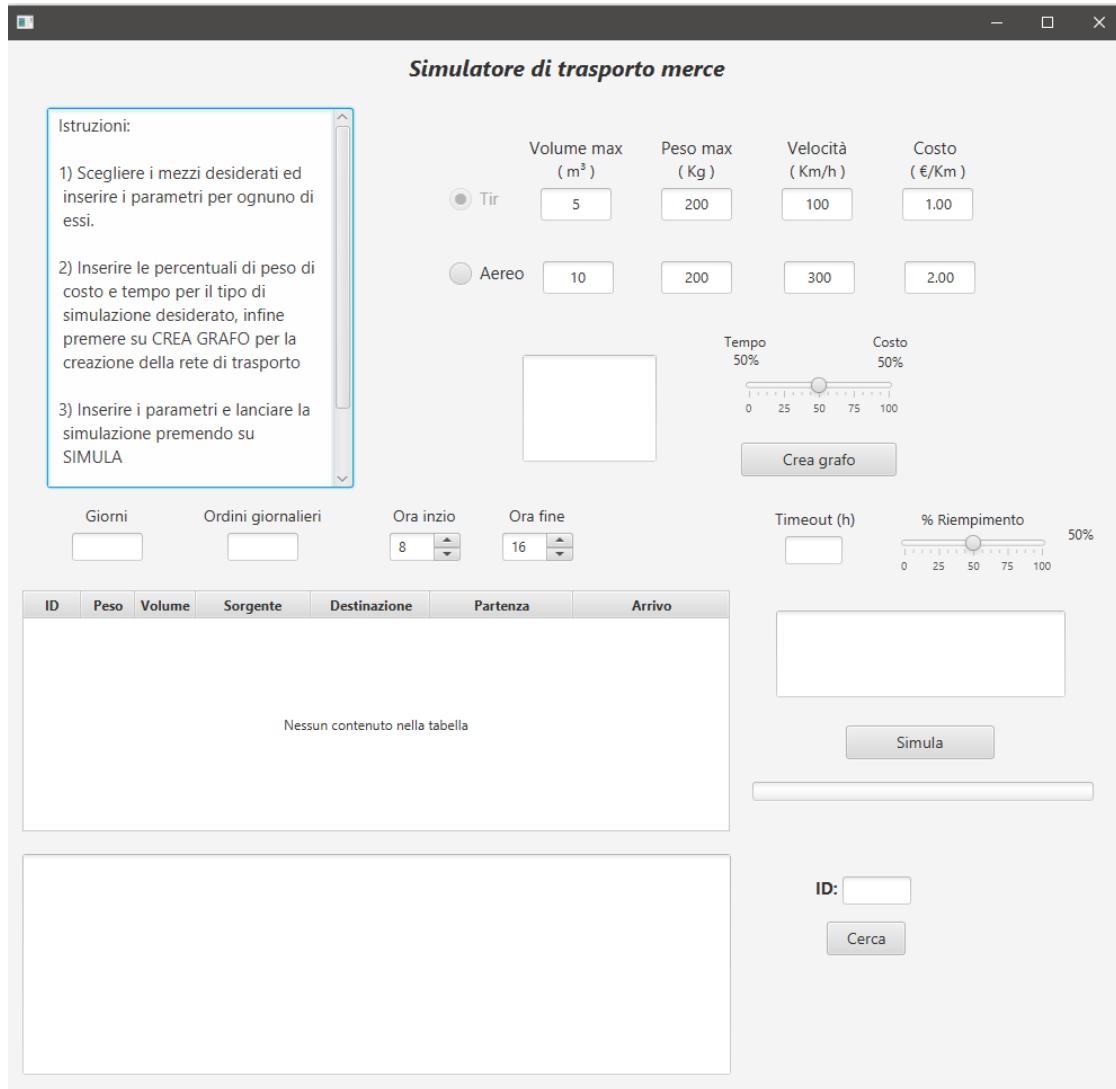
    output.replace("Autobus", "Tir");
    return output;
}

```

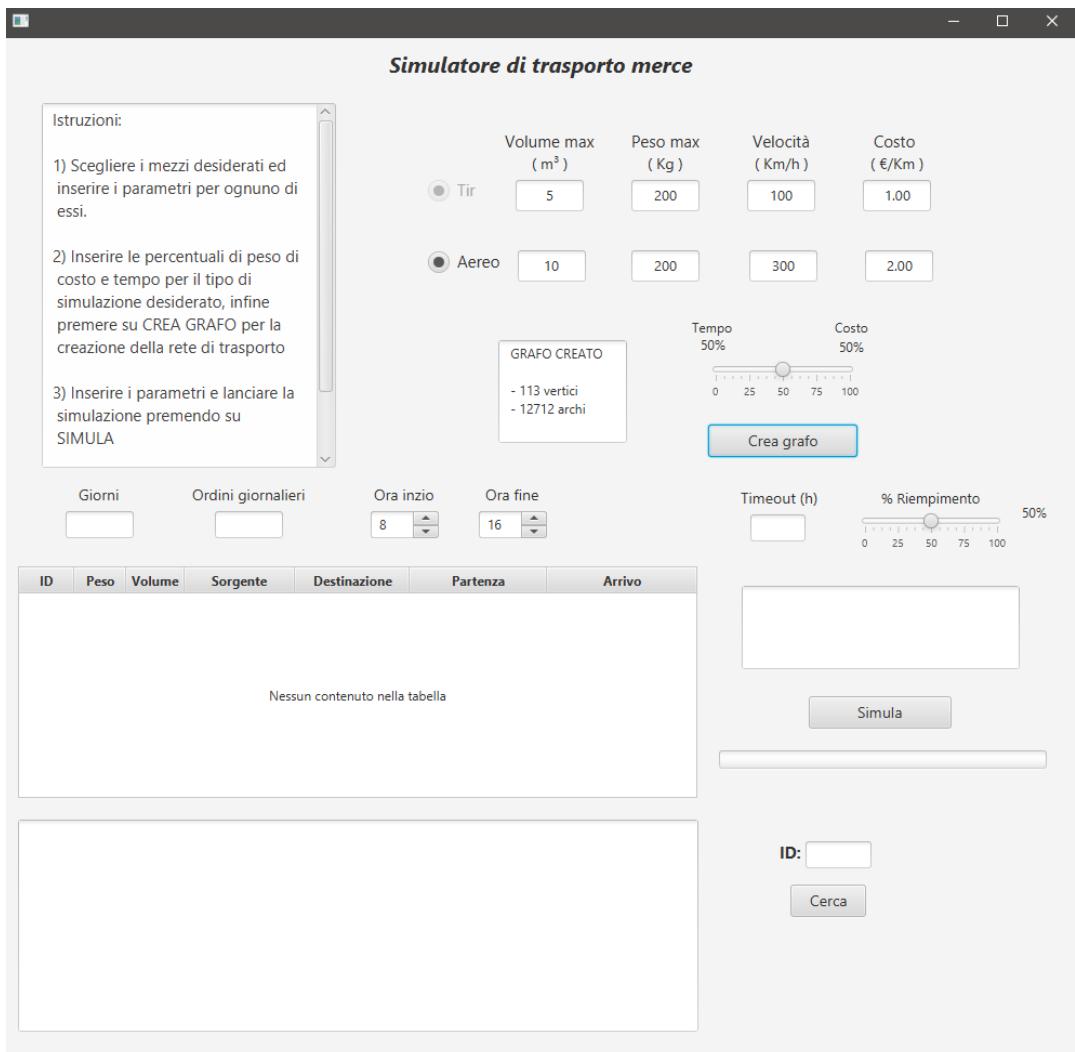
7. Esempi di utilizzo dell'applicazione

All'apertura del software si può notare un campo scorrevole di istruzioni per l'utilizzo dell'applicazione e dei valori caratteristici preimpostati per i mezzi (il tir è il mezzo di default ed è presente sempre); nello spazio sottostante uno slider permette di scegliere la percentuale di peso che devono possedere il tempo e il costo

durante la pesatura degli archi del grafo (di default la pesatura è bilanciata e fa parte del caso in esame).



Cliccando sul bottone “CREA GRAFO” è possibile creare il grafo con le specifiche dell’utente, in output apparirà il messaggio di avvenuta creazione.

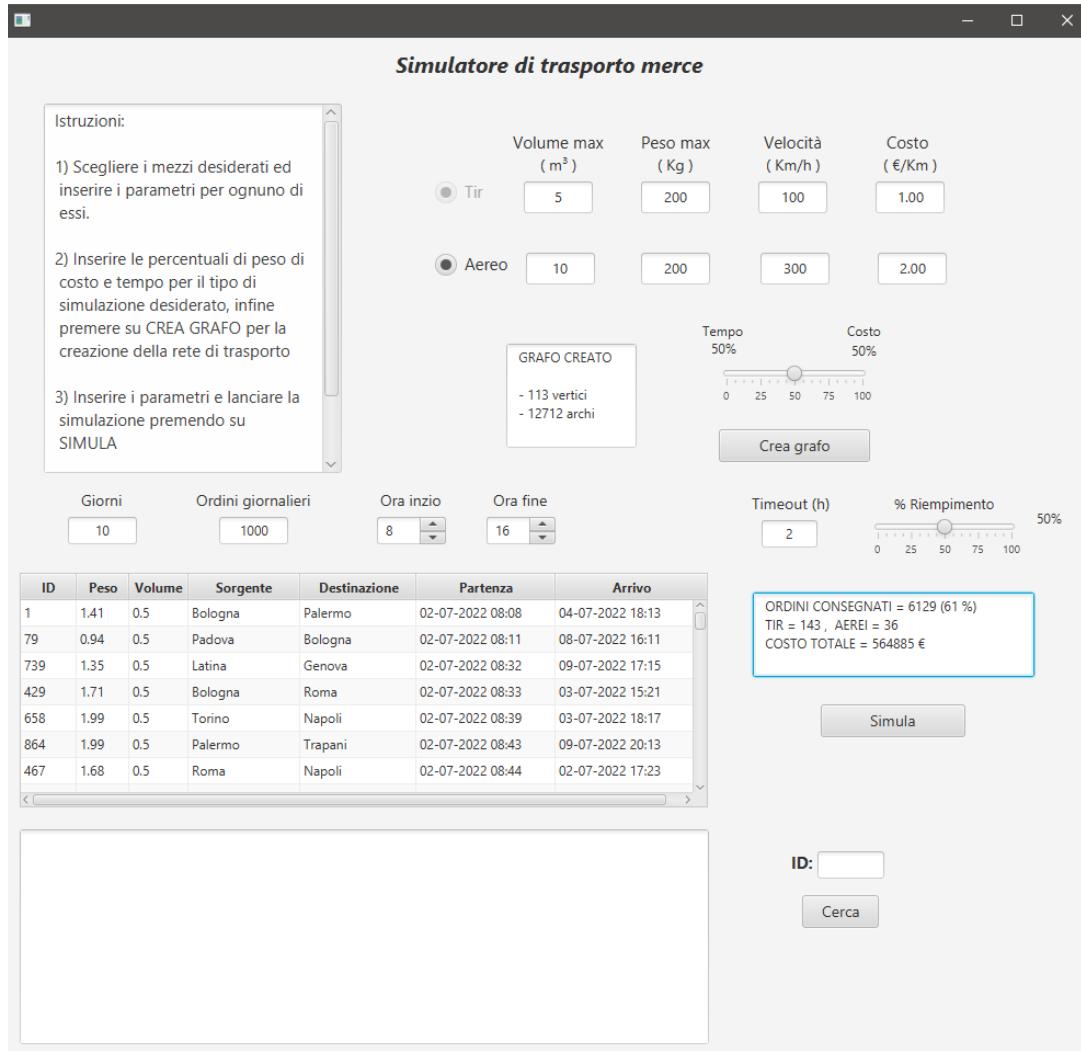


Il passo successivo consiste nell'inserire i parametri della simulazione ed avviarla premendo sul bottone "SIMULA" (è possibile seguire l'andamento della simulazione attraverso la barra di progresso sottostante).

Una volta che la simulazione è terminata si potrà osservare un output generale che riporterà il numero di ordini completati ed il numero di mezzi generati per ogni tipo. Sulla sinistra, una tabella presenterà gli ordini simulati e l'informazione se essi sono stati consegnati o meno (se la data di arrivo è nulla essi sono ancora in processamento all'istante finale).

NOTA: l'applicazione funziona anche per una quantità di ordini fino a 10.000, ma il tempo di processamento può essere anche maggiore di 1 minuto. Fino a 1.000 ordini giornalieri in un numero

di giorni minore di 10 il tempo richiesto è molto minore. In questo esempio viene svolta una simulazione con 1000 ordini giornalieri in 10 giorni e vengono riportati i tempi di computazione per diverse simulazioni.



Dopo la visione degli output della simulazione è possibile inserire l'ID di un ordine (presente nella tabella degli ordini) e cliccare sul bottone "CERCA" per poter osservare sulla sinistra uno storico delle tappe che ha fatto nelle varie città, i mezzi che lo hanno trasportato e le date di ogni tappa (di partenza e di arrivo).

Scorrendo in basso nell'area di testo è possibile vedere il confronto tra il miglior percorso (quello intrapreso) ed il secondo miglior percorso dalla sua origine alla sua destinazione; vengono presentati

inoltre per i 2 percorsi: il peso complessivo del percorso nel grafo, il costo in euro complessivo ed il tempo impiegato in minuti.

Di seguito vengono presentate ricerche di diversi ordini ed i tempi di processamento di diverse simulazioni:

1) ORDINE 36 Milano -> Torino

ID	Peso	Volume	Sorgente	Destinazione	Partenza	Arrivo
872	0.78	0.5	Bologna	Napoli	02-07-2022 09:41	03-07-2022 14:36
36	1.64	0.5	Milano	Torino	02-07-2022 09:42	02-07-2022 16:58
288	1.12	0.5	Bologna	Roma	02-07-2022 09:43	03-07-2022 15:21
606	1.25	0.5	Palermo	Bologna	02-07-2022 09:44	02-07-2022 14:58
878	1.48	0.5	Bologna	Vicenza	02-07-2022 09:44	10-07-2022 19:46
842	1.54	0.5	Como	Bologna	02-07-2022 09:45	05-07-2022 17:38
899	1.47	0.5	Palermo	Milano	02-07-2022 09:48	02-07-2022 22:44

ORDINI CONSEGNATI = 6129 (61 %)
TIR = 143 , AEREI = 36
COSTO TOTALE = 564885 €

Simula

STORICO
Milano Mezzo: 6535 Tir Data: 02-07-2022 15:34
Torino Mezzo: 6535 Tir Data: 02-07-2022 16:58

PERCORSO MIGLIORE:
Milano - Torino mezzo=Tir id=6535

PESO: 1410.59 COSTO: 141.06 € DURATA: 84 minuti

PERCORSO MIGLIORE:
Milano - Torino mezzo=Tir id=6535

PESO: 1410.59 COSTO: 141.06 € DURATA: 84 minuti

PERCORSO ALTERNATIVO:
Milano - Torino mezzo=Aereo id=17

PESO: 1472.57 COSTO: 252.44 € DURATA: 25 minuti

2) ORDINE 400 Catania -> Verbania

ID	Peso	Volume	Sorgente	Destinazione	Partenza	Arrivo
615	0.71	0.5	Napoli	Torino	02-07-2022 10:10	03-07-2022 13:00
207	1.57	0.5	Torino	Genova	02-07-2022 10:14	02-07-2022 14:33
400	1.47	0.5	Catania	Verbania	02-07-2022 10:14	08-07-2022 14:49
947	1.61	0.5	Palermo	Bologna	02-07-2022 10:15	02-07-2022 14:58
34	1.94	0.5	Firenze	Roma	02-07-2022 10:20	03-07-2022 14:12
493	1.17	0.5	Napoli	Bologna	02-07-2022 10:20	02-07-2022 14:58
629	1.07	0.5	Barletta	Roma	02-07-2022 10:20	09-07-2022 20:42
212	1.10	0.5	Genova	Bologna	02-07-2022 10:21	02-07-2022 14:26

ORDINI CONSEGNATI = 6129 (61 %)
TIR = 143 , AEREI = 36
COSTO TOTALE = 564885 €

Simula

STORICO
Catania Mezzo: 2810 Tir Data: 05-07-2022 10:52
Palermo Mezzo: 2810 Tir Data: 05-07-2022 12:57
Palermo Mezzo: 56 Aereo Data: 05-07-2022 14:57
Napoli Mezzo: 56 Aereo Data: 05-07-2022 14:57
Napoli Mezzo: 48 Aereo Data: 05-07-2022 19:44
Bologna Mezzo: 48 Aereo Data: 05-07-2022 21:18
Bologna Mezzo: 1793 Tir Data: 06-07-2022 11:26
Milano Mezzo: 1793 Tir Data: 06-07-2022 13:33
Milano Mezzo: 6546 Tir Data: 08-07-2022 13:49
Verbania Mezzo: 6546 Tir Data: 08-07-2022 14:49

PERCORSO MIGLIORE:
 Catania - Palermo mezzo=Tir id=2810
 Palermo - Napoli mezzo=Aereo id=56
 Napoli - Bologna mezzo=Aereo id=48
 Bologna - Milano mezzo=Tir id=1793
 Milano - Verbania mezzo=Tir id=6546

PESO: 10746.73 COSTO: 1468.1 € DURATA: 408 minuti

PERCORSO ALTERNATIVO:
 Catania - Palermo mezzo=Tir id=2810
 Palermo - Napoli mezzo=Aereo id=56
 Napoli - Firenze mezzo=Tir id=6812
 Firenze - Milano mezzo=Aereo id=30
 Milano - Verbania mezzo=Tir id=6546

PESO: 10753.56 COSTO: 1284.62 € DURATA: 519 minuti

3) ORDINE 5 Genova -> Napoli

4)

ID	Peso	Volume	Sorgente	Destinazione	Partenza	Arrivo
517	0.95	0.5	Genova	Palermo	02-07-2022 10:55	03-07-2022 17:37
771	1.32	0.5	Bologna	Milano	02-07-2022 10:55	02-07-2022 15:32
808	0.55	0.5	Torino	Napoli	02-07-2022 10:56	03-07-2022 18:17
963	1.48	0.5	Roma	Milano	02-07-2022 10:56	03-07-2022 15:30
184	1.15	0.5	Bologna	Firenze	02-07-2022 10:57	03-07-2022 14:33
5	1.04	0.5	Genova	Napoli	02-07-2022 10:59	03-07-2022 16:53
513	1.08	0.5	Palermo	Milano	02-07-2022 10:59	02-07-2022 19:06
89	1.73	0.5	Genova	Ancona	02-07-2022 11:00	07-07-2022 23:31

ORDINI CONSEGNATI = 6129 (61 %)
 TIR = 143 , AEREI = 36
 COSTO TOTALE = 564885 €

Simula

ID: 5

Cerca

STORICO
 Genova Mezzo: 27 Aereo Data: 03-07-2022 14:56
 Napoli Mezzo: 27 Aereo Data: 03-07-2022 16:53

PERCORSO MIGLIORE:
 Genova - Napoli mezzo=Aereo id=27

PESO: 6879.13 COSTO: 1179.28 € DURATA: 117 minuti

PERCORSO ALTERNATIVO:

PERCORSO MIGLIORE:
 Genova - Napoli mezzo=Aereo id=27

PESO: 6879.13 COSTO: 1179.28 € DURATA: 117 minuti

PERCORSO ALTERNATIVO:

Genova - Roma mezzo=Aereo id=22
 Roma - Napoli mezzo=Tir id=9301

PESO: 6885.25 COSTO: 1023.83 € DURATA: 211 minuti

5) ORDINE 356 Bolzano-Bozen -> Roma

ID	Peso	Volume	Sorgente	Destinazione	Partenza	Arrivo
356	0.58	0.5	Bolzano - Bozen	Roma	02-07-2022 11:22	09-07-2022 14:55
281	1.99	0.5	Trapani	Napoli	02-07-2022 11:23	06-07-2022 14:58
431	0.66	0.5	Firenze	Roma	02-07-2022 11:23	03-07-2022 14:12
935	0.96	0.5	Palermo	Milano	02-07-2022 11:24	02-07-2022 22:44
459	1.24	0.5	Genova	Milano	02-07-2022 11:26	03-07-2022 14:56
774	0.7	0.5	Catanzaro	Firenze	02-07-2022 11:26	05-07-2022 14:05
865	0.7	0.5	Bologna	Palermo	02-07-2022 11:26	04-07-2022 18:13

STORICO
 Bolzano - Bozen Mezzo: 1864 Tir Data: 08-07-2022 13:28
 Bologna Mezzo: 1864 Tir Data: 08-07-2022 16:17
 Bologna Mezzo: 36 Aereo Data: 09-07-2022 13:55
 Roma Mezzo: 36 Aereo Data: 09-07-2022 14:55

PERCORSO MIGLIORE:
 Bolzano - Bozen - Bologna mezzo=Tir id=1864
 Bologna - Roma mezzo=Aereo id=36

PESO: 6364.38 COSTO: 889.78 € DURATA: 229 minuti

PERCORSO ALTERNATIVO:
 Bolzano - Bozen - Trento mezzo=Tir id=1947
 Trento - Bologna mezzo=Tir id=11160
 Bologna - Roma mezzo=Aereo id=36

PESO: 6400.51 COSTO: 893.39 € DURATA: 232 minuti

Nel caso in questione il tempo di processamento è stato di 3 minuti, 10000 ordini corrispondono al caso limite oltre il quale i tempi crescono esponenzialmente. Si consiglia di simulare con meno di questa quantità di ordini per avere tempi di processo sostanzialmente brevi.

1000 ordini: TEMPO MEDIO = 8 secondi

2500 ordini: TEMPO MEDIO = 15 secondi

5000 ordini: TEMPO MEDIO = 1 minuto

7500 ordini: TEMPO MEDIO = 2 minuti

10000 ordini: TEMPO MEDIO = 3 minuti

(I tempi di processamento sono discussi nella sezione legata alle criticità e le considerazioni finali).

8. Valutazione dei risultati ottenuti e conclusioni

L'applicazione sviluppata è in grado di rappresentare al meglio uno scenario di trasporto di merci con un data-set più contenuto di quello utilizzato. Il data-set si compone di più di 100 città e più di 20000 archi, il che non favorisce la simulazione (basti pensare che quando avviene il controllo dei mezzi viene controllata una mappa che contiene oggetti per ogni tratta interessata dai percorsi degli ordini, il tutto per un massimo assoluto di mezzi pari al numero di archi). Questo rende i tempi di computazione del software esponenziali. Tuttavia in questo caso i risultati sono più apprezzabili con un numero di ordini alto poiché la simulazione è pensata per gestire numerosi ordini in una rete progettata ad hoc.

I risultati possono essere utili ai fini di una gestione dinamica di una rete di trasporto: quando ad esempio un mezzo non può operare e bisogna trovare l'alternativa migliore, oppure quando una strada è troppo affollata (o chiusa) per lavori...). Inoltre il confronto riportato tra due cammini per un ordine giustifica gli obiettivi di minimizzazione imposti.

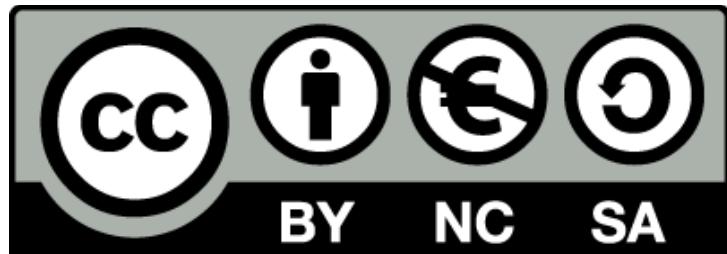
La simulazione restituisce risultati sensati, ma bisogna stare molto attenti agli input immessi. Si consiglia di impostare valori verosimili a dei casi reali (se viene impostato un numero di ordini troppo basso ed una percentuale di riempimento dei mezzi oltre il 50% è ragionevole aspettarsi una minore quantità di mezzi utilizzati).

Poiché un mezzo si occupa di una sola tratta, esso consegna gli ordini e poi torna alla città di partenza (si è deciso di non conteggiare il costo per il ritorno). Sarebbe più sensato che il mezzo attendesse in ogni città in cui giunge prima di ripartire, in modo tale da caricare gli ordini che vanno nella direzione opposta (si ricorda che il grafo è orientato e ad ogni tratta corrispondono 2 archi con direzioni opposte). Si potrebbe pensare di creare il grafo con una logica differente per non generare mezzi sottoutilizzati.

Nella simulazione vengono trascurati i tempi di attesa durante gli scambi degli ordini tra i mezzi, si potrebbe pensare di modellizzare le attese di un ordine rispecchiando casi più realistici.

LINK YOUTUBE DEL VIDEO DIMOSTRATIVO:

<https://youtu.be/5N6vxgUKjaM>



This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

To view a copy of this license, visit

<http://creativecommons.org/licenses/by-nc-sa/4.0/>