



**Politecnico
di Torino**

Politecnico di Torino

Corso di Laurea in Ingegneria Gestionale

Classe L-8

A.A. 2023/2024

Sessione di Laurea Marzo 2024

Tesi di Laurea Triennale

Applicazione Java per la creazione di un itinerario su strada in giro per l'Italia

Relatore:
Professor Fulvio Corno

Candidato:
Aurora Leone, 285107

Sommario

1 - Proposta di progetto	4
1.1 Studente proponente.....	4
1.2 Titolo della proposta.....	4
1.3 Descrizione del problema proposto.....	4
1.4 Descrizione della rilevanza gestionale del problema	4
1.5 Descrizione dei data-set per la valutazione	4
1.6 Descrizione preliminare degli algoritmi coinvolti.....	5
1.7 Descrizione preliminare delle funzionalità previste per l'applicazione software	6
2 - Descrizione dettagliata del problema affrontato.....	8
3 - Descrizione del data-set utilizzato per l'analisi	10
3.1 Tabella delle città.....	10
3.2 Tabella dei percorsi	11
4 - Descrizione ad alto livello delle strutture dati e degli algoritmi utilizzati	12
4.1 Strutture dati utilizzate	12
4.1.1 Package itinerario_italia	12
4.1.2 Package itinerario_italia.db	12
4.1.3 Package itinerario_italia.model	13
4.2 Algoritmi utilizzati.....	14
4.2.1 ItinerarioDAO.....	14
4.2.2 Creazione grafo	15
4.2.3 Algoritmo ricorsivo per trovare l'itinerario migliore	18
5 - Diagramma delle classi delle parti principali dell'applicazione	23
6 - Alcune videate dell'applicazione e link al video	26
7 - Tabelle con risultati sperimentali ottenuti.....	27
7.1 Schermata 1ª modalità di utilizzo.....	27
7.1.1 Esempio 1 scelta delle regioni.....	27
7.1.2 Esempio 2 scelta della zona.....	28
7.1.3 Esempio 3 città di partenza in un'isola	29
7.2 Schermata 2ª modalità di utilizzo.....	30
7.2.1 Esempio 1 scelta delle città	30
8 - Valutazioni sui risultati ottenuti e conclusioni	32

1 - Proposta di progetto

1.1 Studente proponente

s285107 Leone Aurora

1.2 Titolo della proposta

Applicazione Java per la creazione di un itinerario su strada in giro per l'Italia

1.3 Descrizione del problema proposto

L'obiettivo principale è sviluppare un'applicazione dedicata alla pianificazione di viaggi su strada in Italia, offrendo agli utenti un modo efficiente per organizzare gite in giornata o lunghe vacanze in giro per le principali città del Paese.

Gli utenti potranno selezionare la città di partenza, definire un budget, indicare la durata desiderata del soggiorno in ciascuna città e specificare le date e gli orari di partenza e di ritorno.

Inoltre, avranno l'opzione di scegliere se concentrarsi su una o più regioni, preferire una zona geografica dell'Italia (nord, sud, centro) o eventualmente anche esplorare l'intera nazione.

L'utente potrà anche personalizzare ulteriormente l'itinerario, indicando se desidera includere nel proprio itinerario esclusivamente località balneari.

L'applicazione calcolerà automaticamente un itinerario ottimizzato, tenendo conto di tutte le preferenze dell'utente. L'obiettivo è massimizzare il numero di città visitate, rispettando i filtri impostati dall'utente.

Una volta ottenuto un itinerario ottimale, l'utente avrà la possibilità di escludere alcune città dal risultato e ricalcolare un nuovo itinerario, iterando questa operazione finché non si riterrà soddisfatto.

Una variante alternativa dell'applicazione permetterà agli utenti di selezionare le città che intendono visitare. L'applicazione successivamente calcolerà l'itinerario ottimale, mirando a minimizzare le spese complessive del viaggio. Inoltre, tenendo conto della permanenza minima in ogni città, impostata dall'utente, determinerà il tempo minimo necessario per completare l'intero percorso.

1.4 Descrizione della rilevanza gestionale del problema

L'idea è di creare itinerari turistici ottimali basati sulle preferenze individuali dell'utente, rendendo questa applicazione altamente adatta all'uso da parte di tour operator e agenzie di viaggio specializzate nella progettazione di itinerari personalizzati. Potrebbe essere l'applicazione perfetta per gli appassionati di viaggi on the road. Quest'ultima categoria di viaggiatori potrebbe trovare in quest'app un prezioso alleato, specialmente quando si tratta di decidere quali luoghi visitare.

Questa applicazione è particolarmente ideale per gite di qualche giorno, offrendo l'opportunità di scoprire ulteriormente il territorio italiano. La sua capacità di offrire suggerimenti dettagliati può sicuramente incentivare il turismo interno, fornendo agli utenti idee chiare su dove andare, contribuendo a scoprire le più belle città italiane.

1.5 Descrizione dei data-set per la valutazione

Non avendo trovato online un database contenente i dati necessari, dovrà essere creato manualmente utilizzando i dati raccolti da fonti attendibili. Per ottenere i dati sui costi e i tempi di viaggio stimati, si utilizzerà il sito <https://www.viamichelin.com/>, il quale fornisce informazioni sul percorso tra due città. Verranno utilizzati i dati relativi alla durata del tragitto, la distanza in chilometri, il costo del pedaggio e il costo totale.

Il database dovrà essere composto da due tabelle, così strutturate:

TABELLA PERCORSI:

- **IdCollegamento:** campo che rappresenterà l'identificativo univoco del percorso. (integer)
- **IdCittà1:** questo campo conterrà l'identificativo della città di partenza. (integer)
- **IdCittà2:** questo campo conterrà l'identificativo della città di destinazione. (integer)
- **Distanza(km):** qui verrà memorizzata la distanza in chilometri tra le due città. (float)
- **Tempo:** rappresenterà il tempo di percorrenza stimato tra le due città. (varchar hh:mm)
- **Costo_pedaggi:** questo campo conterrà il costo del pedaggio autostradale per il percorso. (float)
- **Costo_totale:** verrà indicato il costo stimato totale per il percorso, include quindi pedaggi e carburante. (double)

TABELLA CITTÀ:

- **IdCittà:** questo campo conterrà il codice univoco della città. (integer)
- **Nome:** questo campo conterrà il nome della città. (varchar)
- **Regione:** indicherà la regione in cui si trova la città. (varchar)
- **Zona:** specifica la zona geografica, ovvero, nord, sud, centro o isola. (varchar)
- **Tipo:** rappresenterà il tipo di località, se è balneare o no. (boolean)
- **Capoluogo:** sarà un valore booleano che indicherà se la città è un capoluogo di regione. (boolean)

L'obiettivo è di includere le principali città per ogni regione, circa quattro per ciascuna di esse, in modo da coprire ampiamente l'intero territorio italiano. In questo modo, gli utenti potranno creare itinerari anche all'interno delle singole regioni, ampliando così le opzioni di viaggio. Rendendo così l'applicazione adatta anche per organizzare brevi gite nella propria regione.

Per facilitarne la creazione, i collegamenti di andata e ritorno verranno considerati equivalenti. Si procederà quindi con l'assunzione che la tratta di andata e quella di ritorno abbiano la stessa durata e lo stesso costo.

1.6 Descrizione preliminare degli algoritmi coinvolti

L'applicazione parte dalla raccolta dei dati inseriti dall'utente e li utilizza per creare un grafo. Ogni vertice di questo grafo rappresenta una città, mentre gli archi tra i vertici rappresentano i collegamenti stradali tra le città. Il grafo comprende la città di partenza specificata dall'utente e tutte le città che soddisfano i filtri impostati dall'utente.

Una volta creato il grafo, l'applicazione applica un algoritmo ricorsivo per determinare l'itinerario ottimale.

Le due modalità di utilizzo dell'applicazione sono spiegate di seguito:

Prima modalità: massimizzazione delle città visitate con rispetto delle preferenze

Nella prima versione, l'itinerario ottimale ottenuto deve rispettare tutte le preferenze dell'utente e massimizzare il numero di città da visitare. In caso di parità di città visitate (ovvero, di archi percorsi), viene selezionato l'itinerario che minimizza il costo totale del viaggio.

L'applicazione suggerisce quindi un itinerario di viaggio, considerando tutte le città nel database che soddisfano i filtri impostati. Una volta ottenuta la combinazione ottimale, l'applicazione offre la possibilità di rimuovere alcune città dal percorso, ad esempio, se l'utente le ha già visitate o per preferenze personali. In questo caso, l'algoritmo viene nuovamente applicato al grafo ridotto, consentendo all'utente di calcolare una nuova soluzione ottimale senza le città non desiderate.

Seconda modalità: selezione manuale delle città da visitare e minimizzazione del costo

Nella seconda versione, l'utente può selezionare manualmente le città da visitare. L'algoritmo troverà quindi la combinazione di città che minimizza i costi di viaggio, identificando l'itinerario più economico e fornendo all'utente l'indicazione dei giorni necessari per completarlo.

In entrambe le versioni, all'utente verrà mostrato l'itinerario completo, il costo totale e la durata complessiva degli spostamenti.

1.7 Descrizione preliminare delle funzionalità previste per l'applicazione software

Prima modalità:

L'applicazione richiede all'utente di fornire una serie di dettagli essenziali per la creazione di un itinerario di viaggio su misura:

- Città di partenza: l'utente specifica la città da cui desidera iniziare il viaggio.
- Budget massimo: l'importo massimo che l'utente è disposto a spendere per l'intero viaggio.
- Data e Orario di partenza: la data e l'orario in cui l'utente prevede di iniziare il viaggio.
- Data e Orario di ritorno: la data e l'orario in cui l'utente prevede di terminare il viaggio.
- Tempo di permanenza in ogni città: i giorni che l'utente desidera trascorrere in ciascuna città visitata.
- Tipo di località (Opzionale): l'utente può specificare se desidera includere esclusivamente località balneari.
- Regioni (Opzionale): la possibilità di selezionare regioni specifiche d'Italia.
- Zona (Opzionale): l'utente può optare per il Nord, il Sud o il Centro d'Italia come area di interesse.

L'applicazione genererà un itinerario di viaggio in grado di soddisfare tutte le preferenze e i requisiti dell'utente, massimizzando il numero di città visitate. Se l'utente non è interessato a visitare alcune delle città consigliate, avrà la possibilità di escluderle dall'itinerario e far riprogettare un nuovo piano, consentendo così all'utente di personalizzare completamente il proprio viaggio.

Seconda modalità:

In alternativa, l'applicazione richiede all'utente di impostare:

- Città di partenza: l'utente specifica la città da cui desidera iniziare il viaggio.
- Tempo di permanenza in ogni città: i giorni che l'utente desidera trascorrere in ciascuna città visitata.
- Elenco di città da visitare: l'utente sceglie una o più città che vorrebbe visitare

In questo caso, l'applicazione fornirà l'itinerario che minimizza il costo e lo comunicherà all'utente insieme al costo complessivo e al tempo necessario per percorrere il percorso selezionato dall'utente.

2 - Descrizione dettagliata del problema affrontato

L'utilità di un'applicazione dedicata alla creazione di itinerari di viaggio deriva dalla crescente complessità e personalizzazione richieste dagli attuali viaggiatori. Il desiderio di viaggiare ed esplorare nuovi luoghi è sempre presente, ma la pianificazione di un viaggio su strada in modo ottimale richiede la considerazione di numerosi fattori, come budget, preferenze personali, limiti temporali e specifiche località da visitare. Un'applicazione specializzata svolge quindi un ruolo cruciale nel semplificare questo processo, fornendo una piattaforma dove gli utenti possono inserire le proprie preferenze e ricevere itinerari personalizzati. L'uso di algoritmi ricorsivi garantisce che gli itinerari siano non solo coerenti con le preferenze dell'utente, ma anche ottimizzati per minimizzare costi e migliorare l'esperienza di viaggio. Un'applicazione di pianificazione dei viaggi diventa dunque uno strumento indispensabile. Elimina la necessità di ricerche laboriose. Ciò è particolarmente importante quando si tratta di gite in giornata o di viaggi più lunghi, dove la gestione efficiente del tempo e delle risorse è essenziale. Inoltre, un'applicazione di questo genere fornisce suggerimenti e proposte che possono ispirare gli utenti a scoprire nuove destinazioni o a esplorare luoghi meno conosciuti. Un'applicazione di pianificazione dei viaggi offre un servizio su misura, adattandosi alle preferenze e ai desideri individuali di ogni utente.

Organizzare un viaggio on the road implica un laborioso processo manuale. L'utente dovrebbe calcolare e valutare manualmente ogni singola possibilità, confrontando diverse opzioni e cercando di massimizzare l'efficienza del loro itinerario. Questo processo non solo richiede un notevole impegno di tempo, ma può anche risultare complesso e soggetto a errori.

L'applicazione proposta emerge come un'innovazione significativa, fornendo un algoritmo che automatizza il processo di pianificazione. L'algoritmo non solo semplifica il lavoro di calcolo, ma offre anche itinerari ottimizzati in base alle preferenze dell'utente. Questa automatizzazione rappresenta una notevole semplificazione del modo in cui i viaggi on the road vengono pianificati, eliminando la necessità di svolgere manualmente complesse analisi e consentendo agli utenti di risparmiare tempo e ottenere un risultato nettamente migliore. Inoltre, il sistema di iterazione permette agli utenti di adattare continuamente i loro piani senza dover ricominciare da capo, migliorando ulteriormente l'efficienza e la personalizzazione del processo di pianificazione.

L'applicazione proposta si impegna dunque ad offrire un servizio di pianificazione di viaggi su strada in Italia, ponendo al centro le esigenze degli utenti e fornendo itinerari ottimizzati che rispecchiano le loro preferenze individuali.

L'applicazione fornisce un'amplia libertà di personalizzazione, gli utenti hanno la libertà di definire la città di partenza, stabilire un budget, specificare la durata desiderata del soggiorno in ogni località, e decidere se concentrarsi su regioni specifiche o una zona geografiche in particolare. Questa flessibilità offre una risposta su misura alle esigenze eterogenee degli utenti, rendendo l'applicazione adatta a una vasta gamma di viaggiatori.

L'uso di un algoritmo ricorsivo garantisce che l'intero itinerario rispetti scrupolosamente tutti i vincoli e le preferenze specificate e che vengano effettivamente valutate tutte le possibili alternative.

Un ulteriore vantaggio offerto dall'applicazione è la flessibilità nell'iterazione. Una volta ottenuto un itinerario ottimale, gli utenti possono escludere alcune città e ricalcolare nuovi itinerari. Questa opzione aggiuntiva conferisce agli utenti la possibilità di migliorare ulteriormente i loro piani di viaggio e rendere così il risultato ottenuto sempre affino alle proprie richieste.

Per i professionisti del settore turistico, come tour operator e agenzie di viaggio specializzate, l'applicazione risulta essere uno strumento strategico. La sua capacità di gestire richieste specifiche degli utenti consente di offrire pacchetti personalizzati e allettanti, contribuendo così a una maggiore soddisfazione del cliente e al successo dell'operatore turistico.

Infine, la possibilità di creare anche gite di breve durata promuove attivamente il turismo interno, incoraggiando gli utenti a scoprire ulteriormente il ricco patrimonio italiano. I suggerimenti forniti dall'applicazione possono generare interesse verso destinazioni meno conosciute, contribuendo così alla promozione e alla valorizzazione delle bellezze delle città italiane.

3 - Descrizione del data-set utilizzato per l'analisi

Come indicato nella proposta, il dataset utilizzato è stato realizzato manualmente, focalizzandosi sulla selezione di alcune delle città principali di ogni regione, approssimativamente tre per ciascuna regione. Attualmente, il dataset comprende un totale di 71 città, con una maggiore rappresentanza di città nelle isole, in quanto queste sono connesse esclusivamente internamente. Questa scelta intenzionale consente di avere una più vasta differenziazione degli itinerari interni alle due isole.

La tabella "città" ha l'identificatore (ID) della città come chiave primaria, fornendo così un riferimento univoco per ogni città presente nel dataset. Dall'altra parte, la tabella "percorsi" contiene tutti i collegamenti possibili tra le città della tabella "città". La chiave primaria della tabella "percorsi" è costituita dall'ID del collegamento, mentre le chiavi secondarie includono gli ID delle due città coinvolte nel collegamento.

Questa organizzazione dei dati fornisce una struttura chiara, consentendo un facile accesso alle informazioni necessarie per la creazione degli itinerari. Di seguito viene mostrata la struttura del dataset.

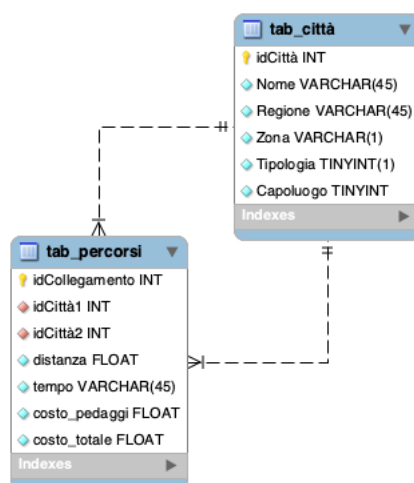


Figura 1 Diagramma ER della struttura del dataset "itinerario_italia"

3.1 Tabella delle città

La tabella "tab_città" racchiude le informazioni sulle città incluse nel dataset. La prima colonna è rappresentata dall'identificativo della città (idCittà) un codice univoco che si incrementa automaticamente con l'aggiunta di nuove città, e che assume il ruolo di chiave primaria.

La struttura della tabella comprende: il campo "Nome" che contiene il nome della località, la colonna "Regione", la quale indica la regione di appartenenza della città. La "Zona" è rappresentata da una singola lettera, che può essere 'N' per il Nord, 'S' per il Sud, 'C' per il Centro e 'I' per le isole ovvero per la Sicilia e per la Sardegna. Il campo "Tipologia" assume il valore 1 se la città è una località balneare, altrimenti 0. Infine, la colonna "Capoluogo" assume il valore 1 se la città è il capoluogo della sua regione, altrimenti 0.

Table: **tab_città**

Columns:

idCittà	int AI PK
Nome	varchar(45)
Regione	varchar(45)
Zona	varchar(1)
Tipologia	tinyint(1)
Capoluogo	tinyint

Figura 2 Schema tabella "tab_città"

Di seguito vengono riportate le prime righe della tabella, mostrando così qualche esempio di dati inseriti.

idCittà	Nome	Regione	Zona	Tipologia	Capoluogo
1	L'Aquila	Abruzzo	S	0	1
2	Pescara	Abruzzo	S	1	0
3	Potenza	Basilicata	S	0	1
4	Matera	Basilicata	S	0	0

Figura 3 Esempio dati inseriti nella tabella "tab_città"

3.2 Tabella dei percorsi

La tabella "tab_percorsi" contiene i dati relativi alle connessioni tra le varie città presenti nella tabella delle città. Il primo campo è l'"idCollegamento", un codice univoco il cui valore cresce automaticamente con l'aggiunta di nuovi collegamenti, il quale svolge la funzione di chiave primaria. All'interno di questa struttura, troviamo i seguenti campi. La coppia di campi "idCittà1" e "idCittà2" identifica le due città coinvolte nel collegamento, il campo "Distanza" fornisce una misurazione della distanza in chilometri tra le città connesse, la rappresentazione temporale è invece affidata al campo "Tempo", espresso nel formato hh:mm, il quale stima la durata prevista del viaggio tra le due città. Il "Costo_pedaggi" rappresenta il costo dei pedaggi autostradali, sono stati sempre selezionati i collegamenti autostradali più veloci rispetto ad alternative secondarie. Infine, il campo "Costo_totale" contiene il costo complessivo del viaggio, unisce quindi i costi dei pedaggi e i costi relativi al carburante. Quest'ultimo valore è calcolato sulla base del consumo stimato di una vettura standard a benzina.

Table: **tab_percorsi**

Columns:

<u>idCollegamento</u>	int AI PK
idCittà1	int
idCittà2	int
distanza	float
tempo	varchar(45)
costo_pedaggi	float
costo_totale	float

Figura 4 Schema tabella "tab_percorsi"

Al fine di agevolare la creazione del database, vista la minima variazione tra i due percorsi, è stata fatta l'assunzione che la tratta di andata e quella di ritorno siano caratterizzate dagli stessi valori.

Di seguito vengono riportate le prime righe della tabella, mostrando così qualche esempio di dati inseriti.

idCollegamento	idCittà1	idCittà2	distanza	tempo	costo_pedaggi	costo_totale
1	1 ➡	2 ➡	100	01:32	7	17.5
2	1 ➡	3 ➡	369	04:29	14.2	54.85
3	1 ➡	4 ➡	417	05:12	22.3	68.3
4	1 ➡	5 ➡	618	06:59	14.2	82.2
5	1 ➡	6 ➡	640	07:29	14.2	91.7
6	1 ➡	7 ➡	705	07:44	14.2	91.7
7	1 ➡	8 ➡	231	03:10	11.7	37.3
8	1 ➡	9 ➡	248	03:17	13.8	41.1
9	1 ➡	10 ➡	280	04:16	13.8	44.9

Figura 5 Esempio dati inseriti nella tabella "tab_percorsi"

4 - Descrizione ad alto livello delle strutture dati e degli algoritmi utilizzati

4.1 Strutture dati utilizzate

Il progetto è stato implementato utilizzando il linguaggio di programmazione Java, seguendo due fondamentali pattern architetturali: il Model-View-Controller (MVC) e il Data-Access-Object (DAO). Questa scelta ha permesso di realizzare un codice modulare, agevolando la gestione e la manutenzione del sistema. A tal fine, sono stati creati tre distinti packages, ognuno svolgente specifiche funzioni, per una chiara suddivisione delle responsabilità all'interno dell'applicazione. Questa struttura organizzativa contribuisce a rendere il codice più comprensibile.

4.1.1 Package `itinerario_italia`

Questo package serve a gestire l'interfaccia utente, presenta la scena all'utente e cattura le sue interazioni. La sua struttura è composta da tre classi distinte:

1. **EntryPoint**: questa classe è responsabile del caricamento iniziale della scena. Si occupa di inizializzare il controller che gestisce la scena, avviare il model e stabilire il collegamento tra il controller e il modello. La classe EntryPoint agisce come punto di ingresso, inizializzando gli elementi chiave necessari per avviare il funzionamento dell'applicazione.
2. **FXMLController**: questa classe ha il compito di raccogliere e gestire tutte le interazioni provenienti dall'utente. Le interazioni raccolte vengono successivamente inoltrate al modello per ottenere risposte specifiche relative al contesto corrente. Inoltre, la classe FXMLController è responsabile dei cambiamenti di stato della scena, garantendo una dinamica interattiva fluida tra l'utente e l'applicazione.
3. **Main**: la classe è fondamentale per l'avvio dell'applicazione. Fornisce il punto di partenza necessario per l'esecuzione del programma, inizializza e mette in moto l'applicazione.

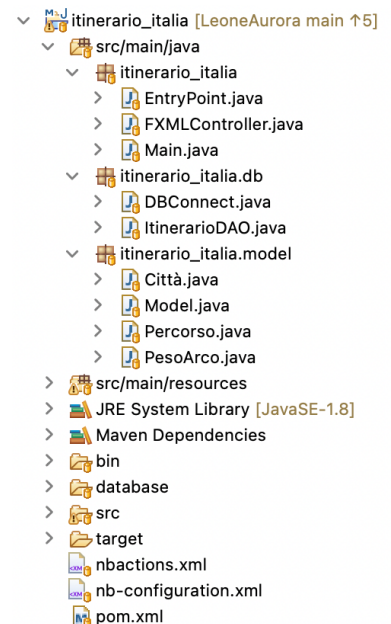


Figura 6 Struttura progetto

4.1.2 Package `itinerario_italia.db`

Questo package serve a stabilire e gestire la connessione con il database e la raccolta dei dati, facilitando il trasferimento dei risultati ottenuti tramite le query al model. È stato appositamente sviluppato per aderire al pattern DAO, garantendo così un accesso limitato e controllato per la lettura e la modifica dei dati. La sua struttura comprende due classi:

1. **ItinerarioDAO**: questa classe svolge la funzione di estrarre dal database tutti i dati desiderati dall'utente mediante l'implementazione di query che estrapolano i dati dal database collegato. Una volta ottenuti i risultati, la classe ItinerarioDAO li passa al modello per ulteriori elaborazioni.

2. **DBConnect**: questa classe è responsabile di stabilire una connessione con il database. Fornisce quindi i mezzi necessari per la gestione delle interazioni tra l'applicazione e il database.

4.1.3 Package `itinerario_italia.model`

Questo package costituisce il nucleo del progetto, poiché tutte le operazioni di elaborazione dei dati sono centralizzate in questo punto. È responsabile della logica applicativa, svolgendo compiti fondamentali per il corretto funzionamento dell'applicazione. La sua struttura è costituita da quattro classi distinte:

1. **Città**: classe implementata per rispettare il pattern **ORM** (Object-Relational Mapping), corrisponde perciò alla tabella delle città. Definisce l'oggetto Città ed ha il proprio costruttore: *Città (idCittà, nome, regione, zona, tipologia, capoluogo)*. È dotata dei relativi getters e setters per ogni attributo della classe, inoltre sono stati implementati i metodi: *hashCode()*, *equals()*, *toString()* e *compareTo()*. Quest'ultimo serve per permettere di ordinare le città in ordine alfabetico.

```
public class Città implements Comparable<Città>{
    private Integer id;
    private String nome;
    private String regione;
    private String zona;
    private Integer tipologia;
    private Integer capoluogo;

    public Città(Integer id, String nome, String regione, String zona, Integer
tipologia, Integer capoluogo) {
        super();
        this.id = id;
        this.nome = nome;
        this.regione = regione;
        this.zona = zona;
        this.tipologia = tipologia;
        this.capoluogo = capoluogo;
    }
}
```

2. **Percorso**: classe implementata per rispettare il pattern **ORM** (Object-Relational Mapping), corrisponde alla tabella contenente i percorsi tra le città. Definisce l'oggetto Percorso ed ha il proprio costruttore: *Percorso (idCittà1, idCittà2, distanza, durata, costoPedaggi, costoTotale)*. Ha i relativi getters e setters per ogni attributo della classe, inoltre sono stati implementati i metodi: *hashCode()*, *equals()* e *toString()*.

```
public class Percorso {
    private Integer id1;
    private Integer id2;
    private Integer distanza;
    private String durata;
    private double costoPedaggi;
    private double costoTot;

    public Percorso(Integer id1, Integer id2, Integer distanza, String durata,
double costoPedaggi, double costoTot) {
        super();
        this.id1 = id1;
        this.id2 = id2;
        this.distanza = distanza;
        this.durata = durata;
        this.costoPedaggi = costoPedaggi;
        this.costoTot = costoTot;
    }
}
```

3. **PesoArco**: classe implementata per creare il peso dell'arco tra due città. Ovvero serve a memorizzare il costo totale e il tempo totale necessario a percorrere l'arco. È dotato di un metodo *convertiDurataInMinuti()* che serve a convertire la stringa della durata dal formato hh:mm a un double che rappresenta i minuti totali.

```
public class PesoArco {
    private double costoTot;
    private double durata;

    public PesoArco(double costoTot, String durata) {
        super();
        this.costoTot = costoTot;
        this.durata = convertiDurataInMinuti(durata);
    }

    private double convertiDurataInMinuti(String durata) {
        String[] tempo = durata.split(":");
        double ore = Double.parseDouble(tempo[0]);
        double minuti = Double.parseDouble(tempo[1]);
        return (ore * 60) + minuti;
    }
}
```

4. **Model**: è il cuore del software, si occupa della gestione di tutti i processi applicativi. Dispone delle strutture di archiviazione dei dati e di tutti i metodi essenziali per elaborare le richieste dell'utente. La sua funzione principale consiste nell'agire come intermediario tra l'applicazione e il database. Contiene tutti codici per la creazione del grafo e l'algoritmo ricorsivo per la creazione dell'itinerario ottimale.

4.2 Algoritmi utilizzati

4.2.1 ItinerarioDAO

Nella classe ItinerarioDao sono contenuti tutti i codici per interrogare il database e ottenere i dati per popolare le combo box dell'interfaccia e per creare il grafo.

public List<Città> getAllCittà() utilizza questa query: `SELECT * FROM tab_città` per ottenere tutte le città presenti nel database

public List<Città> getAllCittàRegione(String regione) utilizza questa query: `SELECT * FROM tab_città WHERE regione=?` per ottenere tutte le città presenti nel database, che si trovano nella regione passata come parametro

public List<Città> getAllCittàZona(String zona) utilizza questa query: `SELECT * FROM tab_città WHERE zona=?` per ottenere tutte le città presenti nel database, che si trovano nella zona passata come parametro

public List<Città> getAllCittàBalneare(Integer balneare) utilizza questa query: `SELECT * FROM tab_città WHERE tipologia=?` per ottenere tutte le città presenti nel database che sono località balneare, ovvero che hanno il parametro tipologia uguale ad 1

public List<String> getAllRegioni() utilizza questa query: `SELECT DISTINCT regione FROM tab_città` per ottenere tutte le regioni

public List<Percorso> getAllpercorsi() utilizza questa query: `SELECT * FROM tab_percorsi` per ottenere tutti i percorsi esistenti nel database

Queste sono usate nel FXMController per popolare le seguenti comboBox presenti nell'interfaccia:

private ComboBox<String> cmbCittà; è la combo box per selezionare la città di partenza

private ComboBox<String> **cmbScegliCittà**; è la combo box per selezionare le città da visitare se si sta utilizzando l'applicazione nella seconda modalità di utilizzo

private ComboBox<String> **cmbRegione**; è la combo box per selezionare le regioni da visitare se si sta utilizzando l'applicazione nella prima modalità di utilizzo

Vengono inoltre utilizzate nel model per creare il grafo in base ai parametri impostati dall'utente.

4.2.2 Creazione grafo

Esistono due metodi per creare il grafo, in base alla modalità di utilizzo dell'applicazione.

Creazione grafo 1ª modalità di utilizzo

```
public List<Città> getCittàVertici(String cittàPartenza, Integer balneare,
List<String> listaRegioni, String zonaScelta) {

    Città cittPartenza = this.cittàIdMap.get(cittàPartenza);

    LinkedList<Città> risultato = new LinkedList<>();

    if (cittPartenza.getRegione().equalsIgnoreCase("Sardegna")) {
        risultato.addAll(this.getAllCittàRegione("Sardegna"));
    } else if (cittPartenza.getRegione().equalsIgnoreCase("Sicilia")) {
        risultato.addAll(this.getAllCittàRegione("Sicilia"));
    } else {
        if (listaRegioni.size() != 0) {
            for (String reg : listaRegioni) {
                risultato.addAll(this.getAllCittàRegione(reg));
            }
        } else if (!zonaScelta.isEmpty()) {
            risultato.addAll(this.getAllCittàZona(zonaScelta));
        } else {
            for (String reg : dao.getAllRegioni()) {
                if (!reg.equals("Sicilia") && !reg.equals("Sardegna")) {
                    risultato.addAll(dao.getAllCittàRegione(reg));
                }
            }
        }
    }

    // Aggiungi la città di partenza solo se non è già presente nella lista
    if (!risultato.contains(cittPartenza)) {
        risultato.add(cittPartenza);
    }

    if (balneare == 1) {
        Iterator<Città> iterator = risultato.iterator();
        while (iterator.hasNext()) {
            Città c = iterator.next();
            if (!this.getAllCittàBalneare(balneare).contains(c) &&
!c.equals(cittPartenza)) {
                iterator.remove();
            }
        }
    }

    return risultato;
}
```

```

public Graph<Città, DefaultEdge> creaGrafo(List<Città> vertici, double budget, double
durataTot, Città partenzaScelta, double permanenza) {

    grafo = new SimpleGraph<>(DefaultEdge.class);
    pesoMap = new HashMap<>();
    grafo.addVertex(partenzaScelta);

    // Lista temporanea per accumulare i vertici da aggiungere dopo
l'iterazione
    List<Città> verticiDaAggiungere = new ArrayList<>();

    // Itera sui percorsi che hanno la città scelta come città di partenza o
arrivo
    dao.getAllpercorsi().stream()
        .filter(percorso ->
            vertici.contains(cittàIdMap2.get(percorso.getId1()))
            && vertici.contains(cittàIdMap2.get(percorso.getId2())))
        .forEach(percorso -> {
            Città cittàPartenza = cittàIdMap2.get(percorso.getId1());
            Città cittàArrivo = cittàIdMap2.get(percorso.getId2());
            double costoTot = percorso.getCostoTot();
            String durata = percorso.getDurata();
            PesoArco pesoArco = new PesoArco(costoTot, durata);

            // Controlla i filtri per l'aggiunta di vertici e archi
            if (costoTot <= budget && pesoArco.getDurata() <= durataTot) {
                if (cittàPartenza.equals(partenzaScelta)) {
                    double costoDoppio = costoTot * 2;
                    double durataCompl =
(pesoArco.getDurata()*2)+permanenza;
                    if (costoDoppio <= budget && durataCompl<= durataTot)
                {
                    verticiDaAggiungere.add(cittàArrivo);
                }
                } else if (cittàArrivo.equals(partenzaScelta)) {
                    double costoDoppio = costoTot * 2;
                    double durataCompl =
(pesoArco.getDurata()*2)+permanenza;
                    if (costoDoppio <= budget && durataCompl<= durataTot)
                {
                    verticiDaAggiungere.add(cittàPartenza);
                }
            }
        });

    // Aggiungi i vertici accumulati al grafo
    verticiDaAggiungere.forEach(grafo::addVertex);

    // Itera sui percorsi rimasti e aggiungi gli archi se entrambe le città
sono vertici del grafo
    dao.getAllpercorsi().stream()
        .filter(percorso ->
            vertici.contains(cittàIdMap2.get(percorso.getId1()))
            &&
            vertici.contains(cittàIdMap2.get(percorso.getId2())))
        .forEach(percorso -> {
            Città cittàPartenza = cittàIdMap2.get(percorso.getId1());
            Città cittàArrivo = cittàIdMap2.get(percorso.getId2());
            double costoTot = percorso.getCostoTot();
            String durata = percorso.getDurata();
            PesoArco pesoArco = new PesoArco(costoTot, durata);

            // Controlla i filtri per l'aggiunta dell'arco
            if (grafo.containsVertex(cittàPartenza) &&
                grafo.containsVertex(cittàArrivo) && costoTot <= budget
                && pesoArco.getDurata() <= durataTot) {

                DefaultEdge arco = grafo.addEdge(cittàPartenza,
cittàArrivo);
                pesoMap.put(arco, pesoArco);
            }
        });

    return grafo;
}

```



```
}
```

Il metodo *getCittàVertici* ha il compito di selezionare le città che costituiranno i vertici del grafo, basandosi sulle scelte dell'utente passate come parametro. Nel caso in cui la città di partenza si trovi in Sicilia o Sardegna, l'utente non ha la possibilità di selezionare specifiche regioni o zone; di conseguenza, automaticamente vengono aggiunte al grafo tutte le città appartenenti alla stessa regione della città di partenza. Se invece l'utente parte da una località al di fuori delle isole, i vertici aggiunti saranno tutte le città appartenenti alle regioni specificate o alla zona selezionata. Se l'utente non ha fatto alcuna scelta riguardo a regioni o zone, il grafo includerà tutte le città presenti nel database.

Il metodo *creaGrafo()* si occupa di aggiungere i vertici e di effettuare un controllo preliminare sugli archi prima di inserirli nel grafo. Viene verificato se il costo complessivo degli archi, considerando sia l'andata che il ritorno dalla città di partenza (quindi moltiplicato per due), supera il budget prestabilito. Allo stesso modo, se la durata eccede il tempo massimo a disposizione dell'utente, il vertice e l'arco corrispondente non vengono aggiunti al grafo. Questa verifica permette di ottimizzare l'esecuzione del programma, creando un grafo già scremato.

Se invece l'arco è idoneo viene aggiunto al grafo e il suo peso, rappresentato da costo totale e durata, viene registrato in una mappa che associa all'arco il relativo peso di classe *PesoArco*.

Creazione grafo 2ª modalità di utilizzo

Per quanto concerne la seconda modalità di utilizzo dell'applicazione, la funzione per creare il grafo è semplificata in quanto l'utente specifica solo la città di partenza, una o più città che vorrebbe visitare e una permanenza media. Non è più necessario avere una funzione per ottenere i vertici in quanto questi sono già scelti dall'utente.

```
public Graph<Città, DefaultEdge> creaGrafo2(List<Città> vertici, Città
partenzaScelta) {
    grafo = new SimpleGraph<>(DefaultEdge.class);
    pesoMap = new HashMap<>();
    grafo.addVertex(partenzaScelta);
    vertici.add(partenzaScelta);

    // Aggiungi i vertici accumulati al grafo
    vertici.forEach(grafo::addVertex);

    // Itera sui percorsi rimasti e aggiungi gli archi se entrambe le città
    sono vertici del grafo
    dao.getAllpercorsi().stream()
        .filter(percorso ->
vertici.contains(cittàIdMap2.get(percorso.getId1())) &&
vertici.contains(cittàIdMap2.get(percorso.getId2())))
        .forEach(percorso -> {
            Città cittàPartenza = cittàIdMap2.get(percorso.getId1());
            Città cittàArrivo = cittàIdMap2.get(percorso.getId2());
            double costoTot = percorso.getCostoTot();
            String durata = percorso.getDurata();
            PesoArco pesoArco = new PesoArco(costoTot, durata);

            // Controlla i filtri per l'aggiunta dell'arco
            if (grafo.containsVertex(cittàPartenza) && g
rafo.containsVertex(cittàArrivo)) {
                DefaultEdge arco = grafo.addEdge(cittàPartenza,
cittàArrivo);
                pesoMap.put(arco, pesoArco);
            }
        })
}
```

```

    });

    return grafo;
}

```

In questo caso, l'utente fornisce direttamente la lista dei vertici come parametro, selezionando le città che intende visitare. In questa fase, non sono più necessari controlli sugli archi, poiché vengono inclusi tutti i collegamenti possibili. Non esistendo limiti di budget o durata, vengono inclusi tutti i potenziali collegamenti nel grafo.

4.2.3 Algoritmo ricorsivo per trovare l'itinerario migliore

4.2.3.1 Algoritmo che massimizza il numero di città visitate e, in caso di parità, minimizza il costo (1ª modalità)

```

public List<DefaultEdge> trovaItinerarioOttimale(Graph<Città, DefaultEdge> grafo,
Città cittàPartenza, double budget, double durataMassima, double permanenza) {

    List<DefaultEdge> migliorItinerario = new ArrayList<>();
    List<DefaultEdge> itinerarioParziale = new ArrayList<>();
    Set<Città> cittàDisponibili = new HashSet<>(grafo.vertexSet());
    int numeroMax = (int) (durataMassima / permanenza);
    numeroMax = Math.min(numeroMax, cittàDisponibili.size());

    cercaItinerarioOttimale(0, numeroMax, grafo, cittàPartenza, cittàPartenza,
itinerarioParziale, migliorItinerario, cittàDisponibili, budget,
durataMassima, permanenza);

    return migliorItinerario;
}

private void cercaItinerarioOttimale(int livello, int numeroMax, Graph<Città,
DefaultEdge> grafo, Città cittàPartenza, Città cittàCorrente, List<DefaultEdge>
itinerarioParziale, List<DefaultEdge> migliorItinerario, Set<Città> cittàDisponibili,
double budget, double durataMassima, double permanenza) {

    if (!itinerarioParziale.isEmpty() &&
isCityInFirstAndLastEdges(itinerarioParziale, cittàPartenza)) {

        double costoMigliorItinerario = calcolaCostoItinerario(migliorItinerario);
        double costoItinerarioParziale = calcolaCostoItinerario(itinerarioParziale);

        if (itinerarioParziale.size() > migliorItinerario.size() ||
(itinerarioParziale.size() == migliorItinerario.size() &&
costoMigliorItinerario > costoItinerarioParziale)) {

            migliorItinerario.clear();
            migliorItinerario.addAll(new ArrayList<>(itinerarioParziale));
        }
    }

    if (livello > numeroMax) {
        return;
    }

    for (DefaultEdge arco : grafo.edgesOf(cittàCorrente)) {
        Città cittàDestinazione = Graphs.getOppositeVertex(grafo, arco,
cittàCorrente);
        if (!cittàDisponibili.contains(cittàDestinazione)) continue; // Ignora città
già visitate

        PesoArco peso = pesoMap.get(arco);
        double costoArco = peso.getCostoTot();
        double durataArco = peso.getDurata() + permanenza;

        if (livello!=0 && (cittàCorrente.equals(cittàPartenza) ||
cittàDestinazione.equals(cittàPartenza))) {
            durataArco = peso.getDurata();
        }
    }
}

```

```

        if (costoArco <= budget && durataArco <= durataMassima) {
            itinerarioParziale.add(arco);
            cittàDisponibili.remove(cittàDestinazione);

            cercaItinerarioOttimale(livello + 1, numeroMax, grafo, cittàPartenza,
            cittàDestinazione, itinerarioParziale, migliorItinerario,
            cittàDisponibili, budget - costoArco, durataMassima - durataArco,
            permanenza);

            itinerarioParziale.remove(arco);
            cittàDisponibili.add(cittàDestinazione);
        }
    }
}

```

L'algoritmo cerca di trovare un itinerario ottimale che massimizzi quindi il numero di città visitate, ovvero di archi percorsi, considerando i vincoli di budget e durata massima. Per prima cosa vengono inizializzate due liste, "migliorItinerario" e "itinerarioParziale", per tenere traccia dell'itinerario ottimale corrente e di quello parziale in esame. L'insieme "cittàDisponibili" contiene tutte le città presenti nel grafo. Viene inoltre calcolato il numero massimo di visite "numeroMax" che è dato dalla durata massima e dal tempo di permanenza in ciascuna città oppure dal numero di vertici del grafo.

Una volta inizializzati tutti i parametri viene chiamata la funzione ricorsiva "cercaItinerarioOttimale". Se l'itinerario parziale contiene almeno un arco e la città di partenza è presente sia all'inizio che alla fine dell'itinerario, si valuta se l'itinerario corrente è migliore di quello attuale in termini di numero di città visitate e costo totale. In caso affermativo, l'itinerario ottimale viene aggiornato. Quando il livello di ricorsione ha raggiunto il numero massimo, la ricorsione termina e torna al livello precedente.

In generale il funzionamento della funzione è il seguente: per ogni arco connesso alla città corrente, si verifica se la città di destinazione è ancora disponibile (non visitata), si calcola quindi il costo e la durata dell'arco corrente, considerando anche il tempo di permanenza nella città di destinazione, se il costo e la durata sono entro i limiti di budget e durata massima, l'arco viene aggiunto all'itinerario parziale e la città di destinazione viene rimossa dalle città disponibili. A questo punto la funzione viene richiamata ricorsivamente con l'itinerario parziale aggiornato e i parametri modificati. Successivamente, l'arco viene rimosso dall'itinerario parziale e la città di destinazione viene ripristinata tra le città disponibili. In questo modo l'algoritmo esplora tutte le possibili combinazioni di percorsi nelle città, tenendo conto dei vincoli di budget e durata massima, e restituisce l'itinerario ottimale in base al numero di città visitate e al costo totale.

Sono inoltre state create alcune funzioni ausiliarie:

```

private boolean isCityInFirstAndLastEdges(List<DefaultEdge> itinerario, Città
cittàPartenza) {
    long count = itinerario.stream()
        .filter(arco -> grafo.getEdgeSource(arco).equals(cittàPartenza) ||
        grafo.getEdgeTarget(arco).equals(cittàPartenza))
        .count();
}

```

```

        if (count == 2) {
            // Verifica se la città compare come primo e ultimo arco
            DefaultEdge primoArco = itinerario.get(0);
            DefaultEdge ultimoArco = itinerario.get(itinerario.size() - 1);

            return (grafo.getEdgeSource(primoArco).equals(cittàPartenza) ||
                    grafo.getEdgeTarget(primoArco).equals(cittàPartenza)) &&
                    (grafo.getEdgeSource(ultimoArco).equals(cittàPartenza) ||
                    grafo.getEdgeTarget(ultimoArco).equals(cittàPartenza));
        }

        return false;
    }

    public double calcolaCostoItinerario(List<DefaultEdge> itinerario) {
        double costoTotale = 0.0;

        for (DefaultEdge arco : itinerario) {
            PesoArco peso = pesoMap.get(arco);
            costoTotale += peso.getCostoTot();
        }

        return Math.round(costoTotale);
    }

    // Metodo ausiliario per calcolare la durata totale dell'itinerario in ore e
    minuti
    public String calcolaDurataTotale(List<DefaultEdge> itinerario) {
        double durataTotaleMinuti = itinerario.stream()
            .mapToDouble(arco -> pesoMap.get(arco).getDurata())
            .sum();

        // Calcola le ore e i minuti
        int ore = (int) (durataTotaleMinuti / 60);
        int minuti = (int) (durataTotaleMinuti % 60);

        if (itinerario.size() == 1) {
            // Calcola le ore e i minuti
            ore = (int) ((durataTotaleMinuti * 2) / 60);
            minuti = (int) ((durataTotaleMinuti * 2) % 60);
        }

        return String.format("%d ore e %d minuti", ore, minuti);
    }
}

```

Le tre funzioni hanno rispettivamente i seguenti scopi: verificare se un itinerario è chiuso, ossia se il primo e l'ultimo arco contengono entrambi la città di partenza come uno dei due vertici, calcolare il costo totale di un itinerario, ed infine calcolare e stampare in un formato comprensibile la durata complessiva dell'itinerario.

4.2.3.2 Algoritmo che minimizza il costo, date le città da includere (2ª modalità)

```

public List<DefaultEdge> trovaItinerarioOttimale2(Graph<Città, DefaultEdge> grafo,
Città cittàPartenza, double permanenza) {
    List<DefaultEdge> migliorItinerario = new ArrayList<>();
    List<DefaultEdge> itinerarioParziale = new ArrayList<>();
    Set<Città> cittàDisponibili = new HashSet<>(grafo.vertexSet());
    int numeroMax = grafo.vertexSet().size();

    cercaItinerarioOttimale2(0, numeroMax, grafo, cittàPartenza, cittàPartenza,
itinerarioParziale, migliorItinerario, cittàDisponibili, permanenza);

    return migliorItinerario;
}

private void cercaItinerarioOttimale2(int livello, int numeroMax, Graph<Città,
DefaultEdge> grafo, Città cittàPartenza, Città cittàCorrente, List<DefaultEdge>

```

```

itinerarioParziale, List<DefaultEdge> migliorItinerario, Set<Città> cittàDisponibili,
double permanenza) {

    if (!itinerarioParziale.isEmpty() &&
        isCityInFirstAndLastEdges(itinerarioParziale, cittàPartenza)) {
        double costoMigliorItinerario = calcolaCostoItinerario(migliorItinerario);
        double costoItinerarioParziale=calcolaCostoItinerario(itinerarioParziale);
        if (itinerarioParziale.size() > migliorItinerario.size() ||
            (itinerarioParziale.size() == migliorItinerario.size() &&
             costoMigliorItinerario > costoItinerarioParziale)) {
            migliorItinerario.clear();
            migliorItinerario.addAll(new ArrayList<>(itinerarioParziale));
        }
    }

    if (livello > numeroMax) {
        return;
    }

    for (DefaultEdge arco : grafo.edgesOf(cittàCorrente)) {
        Città cittàDestinazione = Graphs.getOppositeVertex(grafo, arco,
            cittàCorrente);
        if (!cittàDisponibili.contains(cittàDestinazione)) continue; // Ignora
        città già visitate

        itinerarioParziale.add(arco);
        cittàDisponibili.remove(cittàDestinazione);

        cercaItinerarioOttimale2(livello + 1, numeroMax, grafo, cittàPartenza,
            cittàDestinazione, itinerarioParziale, migliorItinerario,
            cittàDisponibili, permanenza);

        itinerarioParziale.remove(arco);
        cittàDisponibili.add(cittàDestinazione);
    }
}

```

L'algoritmo “trovalitinerarioOttimale2” cerca un itinerario che includa tutte le città che l’utente intende visitare e che minimizzi il costo totale del viaggio.

L’inizializzazione è uguale a quella per la prima modalità di utilizzo, ma in questo caso il numero massimo è dato dal numero di città che l’utente ha inserito, la durata non è più un limite.

La funzione “cercalitinerarioOttimale2”, quando l'itinerario parziale contiene almeno un arco e la città di partenza è presente sia all'inizio che alla fine dell'itinerario, valuta se l'itinerario corrente è migliore di quello ottimale attuale confrontando il costo totale. In caso affermativo, l'itinerario ottimale viene aggiornato.

Se il livello di ricorsione ha raggiunto il numero massimo di visite “numeroMax”, la ricorsione termina e torna al livello precedente. In questo caso non ci sono più controlli sulla durata e sul costo dei singoli archi, perché non ci sono più i vincoli legati al budget e alla durata massima. Quindi per ogni arco che ha la città corrente come vertice, si verifica se la città di destinazione è ancora disponibile (non visitata). In caso positivo l’arco viene aggiunto all'itinerario parziale e la città di destinazione viene rimossa dalle città disponibili. Si procede chiamando ricorsivamente la funzione con l'itinerario parziale aggiornato e i parametri modificati. Successivamente, l'arco viene rimosso dall'itinerario parziale e la città di destinazione viene ripristinata tra le città disponibili.

L'algoritmo esplora dunque tutte le possibili combinazioni di percorsi nelle città e restituisce l'itinerario ottimale ovvero quello che include tutte le città e che ha costo minore.

Inoltre, grazie alla funzione illustrata sotto, all'utente viene mostrato il tempo necessario ad effettuare l'intero percorso.


```
public String calcolaDurataTotaleViaggio(List<DefaultEdge> itinerario, double
permanenza) {
    double durataTotaleMinuti = itinerario.stream()
        .mapToDouble(arco -> pesoMap.get(arco).getDurata())
        .sum();


    double ggTot = (itinerario.size()-1)*permanenza;
    durataTotaleMinuti = durataTotaleMinuti + (ggTot*24*60);
    // Calcola le ore e i minuti
    int giorni = (int) (durataTotaleMinuti / (60*24));
    int ore = (int) ((durataTotaleMinuti % (60 * 24)) / 60);
    int minuti = (int) (durataTotaleMinuti % 60);

    if (itinerario.size()==1) {
        // Calcola le ore e i minuti
        durataTotaleMinuti = (durataTotaleMinuti*2) + (permanenza*24*60);
        giorni = (int) ((durataTotaleMinuti/(60*24)));
        ore = (int) ((durataTotaleMinuti % (60 * 24)) / 60);
        minuti = (int) (durataTotaleMinuti % 60);
    }
}
```

5 - Diagramma delle classi delle parti principali dell'applicazione


Classe Città:

▼  Città.java

▼  Città

- ▣ capoluogo
- ▣ id
- ▣ nome
- ▣ regione
- ▣ tipologia
- ▣ zona
- Città(Integer, String, String, String, Integer, Integer)
- compareTo(Città) : int
- equals(Object) : boolean
- getId() : Integer
- getNome() : String
- getRegione() : String
- getZona() : String
- hashCode() : int
- isCapoluogo() : Integer
- isTipologia() : Integer
- setCapoluogo(Integer) : void
- setId(Integer) : void
- setNome(String) : void
- setRegione(String) : void
- setTipologia(Integer) : void
- setZona(String) : void
- toString() : String

Classe Percorso:

▼  Percorso

- ▣ costoPedaggi
- ▣ costoTot
- ▣ distanza
- ▣ durata
- ▣ id1
- ▣ id2
- Percorso(Integer, Integer, Integer, String, double, double)
- equals(Object) : boolean
- getCostoPedaggi() : double
- getCostoTot() : double
- getDistanza() : Integer
- getDurata() : String
- getId1() : Integer
- getId2() : Integer
- hashCode() : int
- setCostoPedaggi(double) : void
- setCostoTot(double) : void
- setDistanza(Integer) : void
- setDurata(String) : void
- setId1(Integer) : void
- setId2(Integer) : void
- toString() : String

Classe ItinerarioDAO:

- ▼ ItinerarioDAO.java
 - ▼ ItinerarioDAO
 - getAllCittà() : List<Città>
 - getAllCittàBalneare(Integer) : List<Città>
 - getAllCittàRegione(String) : List<Città>
 - getAllCittàZona(String) : List<Città>
 - getAllpercorsi() : List<Percorso>
 - getAllRegioni() : List<String>

Classe Model:

- ▼ Model.java
 - ▼ Model
 - cittàIdMap
 - cittàIdMap2
 - ▣ dao
 - ▣ grafo
 - pesoMap
 - Model()
 - calcolaCostoItinerario(List<DefaultEdge>) : double
 - calcolaDurataTotale(List<DefaultEdge>) : String
 - calcolaDurataTotaleViaggio(List<DefaultEdge>, double) : String
 - ▣ cercaltinerarioOttimale(int, int, Graph<Città, DefaultEdge>, Città, Città, List<DefaultEdge>, List<DefaultEdge>, Set<Città>, double, double, double) :
 - ▣ cercaltinerarioOttimale2(int, int, Graph<Città, DefaultEdge>, Città, Città, List<DefaultEdge>, List<DefaultEdge>, Set<Città>, double) : void
 - creaGrafo(List<Città>, double, double, Città, double) : Graph<Città, DefaultEdge>
 - creaGrafo2(List<Città>, Città) : Graph<Città, DefaultEdge>
 - getAllCittàBalneare(Integer) : List<Città>
 - getAllCittàRegione(String) : List<Città>
 - getAllCittàZona(String) : List<Città>
 - getAllpercorsi() : List<Percorso>
 - getCittà() : List<Città>
 - getCittàVertici(String, Integer, List<String>, String) : List<Città>
 - getNArchi(Graph<Città, DefaultEdge>) : int
 - getNVertici(Graph<Città, DefaultEdge>) : int
 - getRegione() : List<String>
 - ▣ isCityInFirstAndLastEdges(List<DefaultEdge>, Città) : boolean
 - trovaltinerarioOttimale(Graph<Città, DefaultEdge>, Città, double, double, double) : List<DefaultEdge>
 - trovaltinerarioOttimale2(Graph<Città, DefaultEdge>, Città, double) : List<DefaultEdge>

Classe FXMLController:

- ▼  FXMLController.java
 - ▼  FXMLController
 - ▣ btnCalcola
 - ▣ btnCittàScelta
 - ▣ btnConfermaCittà
 - ▣ btnEliminaCittàScelta
 - ▣ btnEliminaREg
 - ▣ btnEliminaZona
 - ▣ btnInviaEscludere
 - ▣ btnInviaRegione
 - ▣ btnInviaZona
 - ▣ btnReset
 - ▣ btnRicalcola
 - ▣ btnScegliCittà
 - ▣ budget
 - ▣ checkBalneare
 - ▣ cittàPartenza
 - ▣ cmbCittà
 - ▣ cmbEscludere
 - ▣ cmbFiltri
 - ▣ cmbRegione
 - ▣ cmbScegliCittà
 - ▣ cmbZona
 - ▣ costoTot
 - ▣ dataPartenza
 - ▣ dataRitorno
 - ▣ grafo
 - ▣ itinerario
 - ▣ listaCittàScelte
 - ▣ listaRegioni
 - ▣ location
 - ▣ modalità
 - ▣ model
 - ▣ orario
 - ▣ orario2
 - ▣ permanenza
 - ▣ permanenzaValore
 - ▣ resources
 - ▣ tempoFinaleM
 - ▣ txtBudget
 - ▣ txtOrario
 - ▣ txtOrario2
 - ▣ txtPermanenza
 - ▣ txtRisultato1
 - ▣ txtRisultato2
 - ▣ txtRisultato3
 - ▣ zonaScelta

- ▲ calcolaltinerario(ActionEvent) : void
- ▲ confermaCittà(ActionEvent) : void
- ▲ eliminaCittàScelta(ActionEvent) : void
- ▲ eliminaReg(ActionEvent) : void
- ▲ eliminaZona(ActionEvent) : void
- ▣ formatOrario(double) : String
- ▲ initialize() : void
- ▲ inviaCittàScelta(ActionEvent) : void
- ▲ inviaEscludere(ActionEvent) : void
- ▲ inviaRegione(ActionEvent) : void
- ▲ inviaZona(ActionEvent) : void
- ▲ resetCampi(ActionEvent) : void
- ▲ ricalcola(ActionEvent) : void
- ▣ sbloccaFiltri() : void
- ▲ scegliCittà(ActionEvent) : void
- > ● setModel(Model) : void
- ▣ updateCmbScegliCittà(String) : void

6 - Alcune videate dell'applicazione e link al video

Come si può vedere dallo schema nella pagina precedente, la classe "FXMLController" gestisce le interazioni con l'utente attraverso una serie di metodi attivati quando l'utente preme i bottoni presenti nell'interfaccia. Il metodo principale è "calcolaitinerario", questo consente all'utente di ottenere il calcolo dell'itinerario ottimale. Tutti i dati richiesti all'utente vengono attentamente verificati per garantire che siano correttamente inseriti e che siano nel formato richiesto.

Per prevenire il blocco dell'applicazione nel caso in cui la ricorsione diventi particolarmente lenta e onerosa, è stato introdotto un timeout impostato a 40 secondi. In caso di superamento del timeout, l'utente viene invitato a modificare alcuni parametri, come ridurre il budget, la durata del viaggio o il numero di regioni visitate.

L'applicazione, essendo progettata per due modalità d'uso, blocca le parti dell'interfaccia non pertinenti alla modalità scelta. Per effettuare nuovamente la scelta della modalità, l'utente deve premere il pulsante "Reset", riportando l'interfaccia alla situazione di partenza. La prima modalità di utilizzo è quella predefinita, mentre per attivare la seconda modalità, l'utente deve premere il pulsante "Scegli città".

Inoltre, la parte dell'interfaccia legata all'eliminazione di alcune città dall'itinerario ottimale, è legata solo alla prima modalità di utilizzo e viene sbloccata solo quando è già stato calcolato un itinerario. In questo caso, vengono bloccati tutti i campi in cui l'utente seleziona i filtri da impostare, dato che devono rimanere bloccati durante il ricalcolo dell'itinerario.

CREA IL TUO VIAGGIO IN GIRO PER L'ITALIA

Città di partenza: Conferma

Budget:

Data di partenza: Data di ritorno:

Orario di partenza: Orario di ritorno:

Tempo di permanenza di ogni città:

Filtri opzionali:

Regioni: Invia Elimina

Zona: Invia Elimina

☐ solo località balneari

Città da escludere: Invia

Figura 7 Interfaccia grafica

LINK per il video illustrativo: <https://www.youtube.com/watch?v=ZBR7NMxk6ww>

7 - Tabelle con risultati sperimentali ottenuti

7.1 Schermata 1ª modalità di utilizzo

7.1.1 Esempio 1 scelta delle regioni

CREA IL TUO VIAGGIO IN GIRO PER L'ITALIA

Città di partenza: Se vuoi trovare un itinerario ottimale scegliendo solo le città da visitare premi qui

Budget:

Data di partenza: Data di ritorno:

Orario di partenza: Orario di ritorno:

Tempo di permanenza di ogni città:

Filtri opzionali:

Regioni:

Zona: ☐ solo località balneari

Stai selezionando le seguenti regioni:
Friuli-Venezia-Giulia
Lazio
Liguria
Toscana

Città visitabili: 13
Torino -> Genova -> Portofino -> Cinque terre -> Pisa -> Firenze -> San Gimignano -> Torino
Costo totale: 169.0
Durata complessiva degli spostamenti: 13 ore e 5 minuti

Città da escludere:

Nella prima schermata, l'applicazione calcola l'itinerario ottimale in base ai parametri forniti. In questo caso, l'itinerario comprende 6 città, limite imposto dalla durata del viaggio selezionata. Tuttavia, si può notare che le città disponibili sono 13, quindi eliminando alcune città potremmo comunque ottenere itinerari sempre comprendenti 6 città, ma che avranno costo maggiore.

Nella schermata successiva, rimuoviamo una delle città dall'itinerario. Ovviamente, essendo che l'itinerario non comprende tutte le città disponibili, eliminando una città, l'algoritmo genera un nuovo itinerario della stessa dimensione, ma con un costo complessivo maggiore. Questo sottolinea il fatto che l'algoritmo, quando si confronta con soluzioni di uguale dimensione, privilegia quella con il costo minore.

CREA IL TUO VIAGGIO IN GIRO PER L'ITALIA

Città di partenza: Se vuoi trovare un itinerario ottimale scegliendo solo le città da visitare premi qui

Budget:

Data di partenza: Data di ritorno:

Orario di partenza: Orario di ritorno:

Tempo di permanenza di ogni città:

Filtri opzionali:

Regioni:

Zona: ☐ solo località balneari

Hai rimosso Portofino dal percorso.

Torino -> Genova -> Cinque terre -> Pisa -> Firenze -> Siena -> San Gimignano -> Torino
Costo totale: 173.0
Durata complessiva degli spostamenti: 13 ore e 25 minuti

Città da escludere:

L'utente può continuare a rimuovere le città che non è interessato a visitare, come si vede nelle due immagini successive.

Hai rimosso Pisa dal percorso.	Torino -> Genova -> Cinque terre -> Firenze -> Siena -> San Gimignano -> Sanremo -> Torino Costo totale: 216.0 Durata complessiva degli spostamenti: 28 ore e 58 minuti		
Città da escludere	<input type="text"/>	<input type="button" value="Invia"/>	<input type="button" value="RICALCOLA ITINERARIO"/>

Hai rimosso Viterbo dal percorso.	Torino -> Trieste -> Firenze -> San Gimignano -> Roma -> Torino Costo totale: 212.0 Durata complessiva degli spostamenti: 19 ore e 31 minuti		
Città da escludere	<input type="text"/>	<input type="button" value="Invia"/>	<input type="button" value="RICALCOLA ITINERARIO"/>

Quando rimane un'unica città disponibile i bottoni per eliminare le città dall'itinerario vengono disabilitati perché non è più possibile creare itinerari differenti; quindi, l'utente dovrà premere il tasto "reset" per poter ricominciare la sua ricerca dell'itinerario perfetto.

Hai rimosso Siena dal percorso.	Torino -> Roma -> Torino Costo totale: 212.0 Durata complessiva degli spostamenti: 9 ore e 46 minuti		
Città da escludere	Siena <input type="text"/>	<input type="button" value="Invia"/>	<input type="button" value="RICALCOLA ITINERARIO"/>

7.1.2 Esempio 2 scelta della zona

CREA IL TUO VIAGGIO IN GIRO PER L'ITALIA

Città di partenza:

Budget:

Data di partenza:
Data di ritorno:

Orario di partenza:
Orario di ritorno:
Tempo di permanenza di ogni città:

Filtri opzionali:

Zona:

Regioni:

Zona:
☐ solo località balneari

Stai selezionando la seguente zona: S	Città visitabili: 18 Roma -> L'Aquila -> Isernia -> Campobasso -> Caserta -> Roma Costo totale: 94.0 Durata complessiva degli spostamenti: 9 ore e 15 minuti		
Città da escludere	<input type="text"/>	<input type="button" value="Invia"/>	<input type="button" value="RICALCOLA ITINERARIO"/>

Hai rimosso Isernia dal percorso.	Roma -> Napoli -> Pompei -> Sorrento -> Caserta -> Roma Costo totale: 96.0 Durata complessiva degli spostamenti: 7 ore e 0 minuti	
Città da escludere	<input type="text"/>	<input type="button" value="Invia"/> <input type="button" value="RICALCOLA ITINERARIO"/>

7.1.3 Esempio 3 città di partenza in un'isola

CREA IL TUO VIAGGIO IN GIRO PER L'ITALIA

Città di partenza: Se vuoi trovare un itinerario ottimale scegliendo solo le città da visitare premi qui

Budget: Scegli le città da visitare:

Data di partenza:

Orario di partenza:

Orario di ritorno:

Tempo di permanenza di ogni città:

Filtri opzionali:

Regioni:

Zona: ☐ solo località balneari

	Città visitabili: 7 Agrigento -> Palermo -> Cefalù -> Noto -> Siracusa -> Catania -> Agrigento Costo totale: 90.0 Durata complessiva degli spostamenti: 10 ore e 10 minuti	
Città da escludere	<input type="text"/>	<input type="button" value="Invia"/> <input type="button" value="RICALCOLA ITINERARIO"/>

Essendo la città di partenza nella regione Sicilia, l'itinerario può essere solo interno quindi l'utente non è abilitato a scegliere le regioni o la zona da visitare. In questo caso si vede che l'itinerario non include tutte le città disponibili perché sfiorerebbe il budget impostato.

7.2 Schermata 2ª modalità di utilizzo

7.2.1 Esempio 1 scelta delle città

CREA IL TUO VIAGGIO IN GIRO PER L'ITALIA

Città di partenza: Venezia Se vuoi trovare un itinerario ottimale scegliendo solo le città da visitare premi qui

Budget:

Data di partenza: Data di ritorno:

Orario di partenza:

Orario di ritorno:

Tempo di permanenza di ogni città:

Filtri opzionali:

Regioni:

Zona: ☐ solo località balneari

Città da visitare:

Genova
Gressoney
Isernia
Lecce
L'Aquila
Mantova
Matera
Milano
Modena
Napoli

Città da escludere:

Alla pressione del pulsante "Scegli città", tutte le funzionalità legate alla prima modalità di utilizzo vengono disabilitate, mentre quelle associate alla seconda modalità vengono attivate. La combo box per selezionare le città da visitare viene compilata con l'elenco completo delle città disponibili nel database. Nel caso in cui la città di partenza selezionata si trovi su una delle due isole, la scelta sarà limitata alle città appartenenti a quella specifica isola.

CREA IL TUO VIAGGIO IN GIRO PER L'ITALIA

Città di partenza: Venezia Se vuoi trovare un itinerario ottimale scegliendo solo le città da visitare premi qui

Budget:

Data di partenza: Data di ritorno:

Orario di partenza:

Orario di ritorno:

Tempo di permanenza di ogni città:

Filtri opzionali:

Regioni:

Zona: ☐ solo località balneari

Città da visitare:

Torino

Stai selezionando le seguenti città:
Como
Firenze
Napoli
Pisa
Roma
Torino

Venezia -> Roma -> Napoli -> Firenze -> Pisa -> Torino -> Como -> Venezia
Costo totale: 328.0
Durata complessiva degli spostamenti: 19 ore e 59 minuti
Tempo necessario: 18 giorni e 19 ore e 59 minuti

Città da escludere:

Una volta scelte le città da visitare, premendo il bottone “calcola itinerario” viene generato l’itinerario ottimo, ovvero quello che include tutte le città scelte ed ha costo minore. Oltre al percorso consigliato vengono stampati il costo totale, la durata complessiva degli spostamenti e la quantità di tempo necessario a completare l’intero percorso in base alla permanenza impostata dall’utente.

CALCOLA ITINERARIO

RESET

Stai selezionando le seguenti città:
Como
Firenze
Napoli
Pisa
Roma
Torino
Trieste

Venezia -> Trieste -> Como -> Torino -> Pisa -> Firenze -> Napoli -> Roma -> Venezia
Costo totale: 385.0
Durata complessiva degli spostamenti: 23 ore e 8 minuti
Tempo necessario: 21 giorni e 23 ore e 8 minuti

Città da escludere

Invia

RICALCOLA ITINERARIO

8 - Valutazioni sui risultati ottenuti e conclusioni

Il cuore dell'applicazione è costituito dai due algoritmi per il calcolo dell'itinerario ottimale, i quali costituiscono la parte più onerosa in termini di risorse utilizzate. I tempi di risposta dipendono dall'input fornito dall'utente.

Ci sono tre aspetti chiave che influenzano la velocità di esecuzione dell'algoritmo utilizzato nella prima modalità di utilizzo dell'applicazione:

Numerosità di vertici e archi del grafo: maggiore è il numero di collegamenti possibili, maggiori saranno le combinazioni da esaminare.

Budget e durata impostati: Un budget e una durata di viaggio meno elevati permettono di escludere alcuni archi prima di eseguire l'algoritmo, alleggerendo così la ricorsione.

Permanenza impostata: Il valore di permanenza influenza la dimensione massima dell'itinerario e, in alcuni casi, accelera il calcolo ricorsivo. Valori elevati di permanenza accelerano il calcolo, riducendo il numero massimo di città visitabili, facendo così terminare prima la ricorsione.

L'algoritmo relativo alla prima modalità di utilizzo effettua una ricerca più intensiva, considerando vari fattori e iterando su un maggior numero di combinazioni essendo che l'utente seleziona intere regioni o addirittura un'intera zona della nazione. Il calcolo ricorsivo si alleggerisce man mano che l'utente elimina alcune città dall'itinerario, riducendo le dimensioni del grafo e quindi le combinazioni da calcolare.

Invece, per quando riguarda il secondo algoritmo, legato al secondo metodo di utilizzo, questo è influenzato esclusivamente dalla numerosità di vertici e archi del grafo. La sua complessità è teoricamente alleggerita dal fatto che l'utente seleziona direttamente le città da visitare, evitando di iterare su città che non intende visitare. Se l'utente utilizza l'applicazione in modo consapevole, selezionerà esattamente le città desiderate, limitando il numero di iterazioni rispetto all'utilizzo del primo algoritmo.

In conclusione, l'applicazione è in grado di generare in tempi ragionevoli l'itinerario ottimale se utilizzata correttamente e con dati realistici. L'interruzione del calcolo dell'itinerario dovuto al superamento del limite di tempo, impostato tramite il timeout, solitamente si verifica quando l'utente sceglie un numero di regioni eccessivo, superiore a 4. Allo stesso modo, si può verificare se l'utente tenta di iterare sull'intera nazione senza specificare regioni o se seleziona un'intera zona con un budget particolarmente elevato. Tuttavia, in generale, l'applicazione offre un'esperienza fluida quando utilizzata in modo appropriato.



Quest'opera è distribuita con Licenza [Creative Commons Attribuzione - Non commerciale - Condividi allo stesso modo 4.0 Internazionale](https://creativecommons.org/licenses/by-nc-sa/4.0/).

