- Main

- Grafo

- Dao

- Query (indice di correlazione, lista ordinata)

- Cammino minimo Dijkstra

- Componente connessa

- Scelta random

- Timestamp to string

- Comparatore archi

- Distanza

- Stringa da edge set

- Combo box

- Ricorsione (cammino minimo, massimo)

- Simulazione

- Query DateDiff

NB: CAMBIA PASSWORD NEL DB CONNECT
NEL CONTROLLER BISOGNA ISTANZIARE IL MODEL E I METODI GETTER E SETTER
E POI NEL MAIN PRIMA DI SCENE E DOPO BORDERSCONTROLLER

```
Model model= new Model();
```

MAIN ------ Come dovrebbe essere

```
public class Main extends Application {
	@Override
	public void start(Stage primaryStage) {
		try {
			FXMLLoader loader = new
FXMLLoader(getClass().getResource("PowerOutages.fxml"));
			BorderPane root = (BorderPane) loader.load();
			PowerOutagesController controller = loader.getController();

			Model model = new Model();
			controller.setModel(model);

			Scene scene = new Scene(root);


scene.getStylesheets().add(getClass().getResource("application.css").toExternalForm());
			primaryStage.setScene(scene);
			primaryStage.show();

		} catch (Exception e) {
			e.printStackTrace();
		}
	}
```

GRAFO

-model

```
private UndirectedGraph<Country, DefaultEdge> graph;

public void creaGrafo(int anno) {
		this.graph= new SimpleGraph<>(DefaultEdge.class);

		// add vertex
		BordersDAO dao= new BordersDAO();
		this.countries= dao.loadAllCountries();

		//Mappa identita'

		this.countryMap = new HashMap<>();
		for (Country c: this.countries)
		{
			this.countryMap.put(c.getcCode(),c);
		}

		Graphs.addAllVertices(graph, this.countries);


		// add edges

		List<IntegerPair> confini= dao.getCountryPairs(anno);

		for (IntegerPair p: confini)
			graph.addEdge(this.countryMap.get(p.getN1()),
					this.countryMap.get(p.getN2()));

	}
```

METODI DAO CON PARAMETRI

INTERROGARE DB (NEL DAO)

```
public List<IntegerPair> getCountryPairs (int anno){

		String sql = "SELECT state1no, state2no " +
				"FROM contiguity " +
				"WHERE year<=? " ;

		List <IntegerPair> result= new ArrayList<>();
		Connection conn= DBConnect.getConnection();
		try {
			PreparedStatement st = conn.prepareStatement(sql) ;
			st.setInt(1, anno);

			ResultSet res= st.executeQuery();

			while( res.next()) {
				result.add(new IntegerPair(res.getInt("state1no"),

	res.getInt("state2no")));
			}

		conn.close();
		return result;

		} catch (SQLException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}

		return null;

	}
```

QUERY

INDICE DI CORRELAZIONE

```
SELECT DISTINCT YEAR(p1.date_event_began), MONTH (p1.date_event_began)
FROM poweroutages p1, poweroutages p2
WHERE p1.nerc_id=4
AND p2.nerc_id=3
AND YEAR(p1.date_event_began)=YEAR(p2.date_event_began)
AND MONTH (p1.date_event_began)=MONTH (p2.date_event_began)
```

il count si fa nel DAO con un while

LISTA ORDINATA

```
SELECT distinct(STATE) from airports ORDER BY STATE ASC
```

CAMMINO MINIMO con DIJKSTRA (SOLO UNA VOLTA)

```
public List<Fermata> trovaCamminoMinimo (Fermata partenza, Fermata arrivo){

DijkstraShortestPath<Fermata,DefaultWeightedEdge>dijkstra= new
DijkstraShortestPath<>(this.grafo);

GraphPath<Fermata, DefaultWeightedEdge> path= dijkstra.getPath(partenza,arrivo);
return path.getVertexList():


}
```

TUTTI I CAMMINI MINIMI CON FLOYD

COMPONENTE CONNESSA (num vertici del grafo connesso)

```
	public int calcolaDimensioneCC(int idObj) {

		// trova il vertice di partenza
		ArtObject start = trovaVertice(idObj);

		// visita il grafo
		Set<ArtObject> visitati = new HashSet<>();
		DepthFirstIterator<ArtObject, DefaultWeightedEdge> dfv = new
DepthFirstIterator<>(this.graph, start);
		while (dfv.hasNext())
			visitati.add(dfv.next());

		// conta gli elementi
		return visitati.size();
	}
```

SCELTA RANDOM DA LISTA

```
private int random() {
		double prob = Math.random();
		return (int) (prob*(aeroporti.size()-1));
	}
```

TRASFORMARE TIMESTAMP IN STRING

```
Timestamp ts = Timestamp.valueOf(primoVolo.getArrivalDate());
			String dateok = "";
			for(int i=0; i<ts.toString().length()-2; i++) {
				dateok+=ts.toString().charAt(i);
			}
```

COMPARATORE ARCHI

```
   Collections.sort(outgoing, new Comparator<DefaultWeightedEdge>(){


		@Override

		public int compare(DefaultWeightedEdge o1, DefaultWeightedEdge o2) {

			return (int)graph.getEdgeWeight(o2) - (int)graph.getEdgeWeight(o1);

		}});
```

DISTANZA

```
	private double getDistanza(Airport a1, Airport a2) {


		double distanza = LatLngTool.distance(new LatLng(a1.getLatitude(),
								a1.getLongitude()),
					new LatLng(a2.getLatitude(),a2.getLongitude()),
								LengthUnit.KILOMETER);

		return distanza;
	}
```

STRINGA DI STAMPA CON ROTTE PEGGIORI A PARTIRE DA EDGE SET

```
public String getPeggioriRotte() {
		List<RitardoTratta> rotte = new ArrayList<RitardoTratta>();
		List<RitardoTratta> peggiori = new ArrayList<RitardoTratta>();
		for(DefaultWeightedEdge e : grafo.edgeSet()) {
			int o = grafo.getEdgeSource(e).getId();
			int oNome = airports.get(o).getId();
			int d = grafo.getEdgeTarget(e).getId();
			int dNome = airports.get(d).getId();
			double ritPesato = grafo.getEdgeWeight(e);
			rotte.add(new RitardoTratta(oNome, dNome, ritPesato));
		}
		Collections.sort(rotte);
		String s ="";
		for(int i=0; i<10; i++) {
			peggiori.add(rotte.get(i));
			String a1 = airports.get(rotte.get(i).getA1()).getIata();
			String a2 = airports.get(rotte.get(i).getA2()).getIata();
			s+=a1+" - "+a2+" "+peggiori.get(i).getAvgRitardo()+"\n";
		}
		return s.trim();
	}
```

COMBO BOX

NB inserire la tipologia di oggetti nel controller al ComboBox <?> al posto del punto interrogativo

```
	//popolo la tendina delle country generate

	boxNazione.getItems().clear();

	for (CountryNum c: stati)
	{
		boxNazione.getItems().add(c.getCountry());
	}
```

PER AVERE VALORE SELEZIONATO IN TENDINA: Country partenza= boxNazione.getValue();

RICORSIONE

CAMMINO MINIMO

```
	List<Corso> findMinimalSet(){
		List<Corso> soluzioneParziale = new ArrayList<Corso>();
		List<Corso> soluzioneMigliore = new ArrayList<Corso>();

		recursive(soluzioneParziale, soluzioneMigliore);

		return soluzioneMigliore;
	}

	void recursive(List<Corso> soluzioneParziale, List<Corso> soluzioneMigliore) {

//		System.out.println(soluzioneParziale);

		HashSet<Studente> hashSetStudenti = new
HashSet<Studente>(getTuttiStudenti());
		for (Corso corso : soluzioneParziale) {
			hashSetStudenti.removeAll(corso.getStudenti());
		}
		if (hashSetStudenti.isEmpty()) {
			if (soluzioneMigliore.isEmpty())
				soluzioneMigliore.addAll(soluzioneParziale);
			if (soluzioneParziale.size() < soluzioneMigliore.size()){
				soluzioneMigliore.clear();
				soluzioneMigliore.addAll(soluzioneParziale);
			}
		}

		for (Corso corso : getTuttiCorsi()) {
			if (soluzioneParziale.isEmpty() ||
corso.compareTo(soluzioneParziale.get(soluzioneParziale.size()-1)) > 0) {
				soluzioneParziale.add(corso);
				recursive(soluzioneParziale, soluzioneMigliore);
				soluzioneParziale.remove(corso);
			}
		}

	}
```

CAMMINO PIU' LUNGO

```
//
public List<Season> trovaCamminoVirtuoso() {

		ottima = new ArrayList<>();
		List<Season> parziale= new ArrayList<>();

		y=0;

		parziale.add(punteggiPerStagioni.get(0).getS());
		pv= punteggiPerStagioni.get(0).getPunteggio();
		cerca(parziale);

		return ottima;


	}

	public void cerca(List<Season> parziale) {

		if( parziale.size()>ottima.size())
		{
			this.ottima=new ArrayList<>(parziale);
		}

		int i=0;

		for (AnnoPunteggio s: punteggiPerStagioni)
		{

			if(s.getPunteggio() > pv && i>=y)
			{
				parziale.add(s.getS());
				pv=s.getPunteggio();
				y=i;
				cerca(parziale);

			}

			else if (s.getPunteggio() <= pv && i>y)
			{
				parziale.clear();
				parziale.add(s.getS());
				pv= s.getPunteggio();

			}

			i++;
		}

	}
//
```

```
public List<Classifica> getVirtuosa() {

		this.ottima = new LinkedList<Classifica>();
		List<Classifica> set = new LinkedList<Classifica>(grafo.vertexSet());
		List<Classifica> parziale = new LinkedList<>();
		for(Classifica c : set) {
			parziale = new LinkedList<>();
			parziale.add(c);
			cercaVirtuosa(parziale);
		}



		System.out.println(ottima.toString());

		return this.ottima;

	}

	private void cercaVirtuosa(List<Classifica> parziale) {
		if(parziale.size()>ottima.size())
			this.ottima = new LinkedList<>(parziale);

		List<Classifica> candidati = new
LinkedList<Classifica>(getSuccessivi(parziale.get(parziale.size()-1)));
		for(Classifica c : candidati) {
			if(!parziale.contains(c)) {
				if(parziale.get(parziale.size()-1).getPunti()<c.getPunti() &&
						parziale.get(parziale.size()-
1).getStagione().getSeason()==c.getStagione().getSeason()-1) {
					parziale.add(c);
					cercaVirtuosa(parziale);
				}
			}
		}
	}

	private List<Classifica> getSuccessivi(Classifica classifica) {

		return Graphs.successorListOf(grafo, classifica);
	}
```

CAMMINO MASSIMO

```
public List<ArtObject> camminoMassimo(int startId, int LUN) {
		// trova il vertice di partenza
		ArtObject start = trovaVertice(startId);

		List<ArtObject> parziale = new ArrayList<>();
		parziale.add(start);

		this.best = new ArrayList<>();
		best.add(start);

		cerca(parziale, 1, LUN);

		return best;

	}

	private void cerca(List<ArtObject> parziale, int livello, int LUN) {
		if (livello == LUN) {
			// caso terminale
			if (peso(parziale) > peso(best)) {
				best = new ArrayList<>(parziale);
				System.out.println(parziale);
			}
			return;
		}

		// trova vertici adiacenti all'ultimo
		ArtObject ultimo = parziale.get(parziale.size() - 1);

		List<ArtObject> adiacenti = Graphs.neighborListOf(this.graph, ultimo);

		for (ArtObject prova : adiacenti) {
			if (!parziale.contains(prova) && prova.getClassification() != null
					&&
prova.getClassification().equals(parziale.get(0).getClassification())) {
				parziale.add(prova);
				cerca(parziale, livello + 1, LUN);
				parziale.remove(parziale.size() - 1);
			}
		}

	}

	private int peso(List<ArtObject> parziale) {
		int peso = 0;
		for (int i = 0; i < parziale.size() - 1; i++) {
			DefaultWeightedEdge e = graph.getEdge(parziale.get(i), parziale.get(i +
1));
			int pesoarco = (int) graph.getEdgeWeight(e);
			peso += pesoarco;
		}
		return peso;
	}
```

SIMULAZIONE

-EVENTO

```
public class Evento implements Comparable<Evento>{

	public enum TipoEvento{
		CRIMINE,
		ARRIVA_AGENTE,
		GESTITO
	}

	private TipoEvento tipo;
	private LocalDateTime data;
	private Event crimine;

	public Evento(TipoEvento tipo, LocalDateTime data, Event crimine) {
		super();
		this.tipo = tipo;
		this.data = data;
		this.crimine = crimine;
	}

	public TipoEvento getTipo() {
		return tipo;
	}
	public void setTipo(TipoEvento tipo) {
		this.tipo = tipo;
	}
	public LocalDateTime getData() {
		return data;
	}
	public void setData(LocalDateTime data) {
		this.data = data;
	}
	public Event getCrimine() {
		return crimine;
	}
	public void setCrimine(Event crimine) {
		this.crimine = crimine;
	}

	@Override
	public int compareTo(Evento o) {
		return this.data.compareTo(o.getData());
	}



}
```

-SIMULATORE

```
public class Simulatore {
	//TIPI DI EVENTO
	//1. Evento Criminoso
	//	1.1 La centrale seleziona
	//		l'agente piÃ¹ vicino

	//	1.2 Setta l'agente a occupato

	//2. Arriva agente
	//	2.1 Definisco quanto durerÃ  l'intervento
	//	2.2 Controlla se l'evento
	//		Ã¨ mal gestito
	//3. Crimine terminato
	//	3.1 Libero l'agente

	// Strutture dati che ci servono
	private Integer malGestiti;
	private Integer N;
	private Integer anno;
	private Integer mese;
	private Integer giorno;                                  PARAMETRI INPUT OUTPUT
	private Graph<Integer, DefaultWeightedEdge> grafo;
	private PriorityQueue<Evento> queue;

	//mappa di distretto-# agenti
	private Map<Integer,Integer> agenti;                     MODELLO STATO SISTEMA


	public void init(Integer N, Integer anno, Integer mese, Integer giorno,
			Graph<Integer, DefaultWeightedEdge> grafo) {
		this.N = N;
		this.anno = anno;
		this.mese = mese;
		this.giorno = giorno;
		this.grafo = grafo;

		this.malGestiti = 0;
		this.agenti = new HashMap<Integer,Integer>();
		for(Integer d : this.grafo.vertexSet()) {
			this.agenti.put(d, 0);
		}

		//devo scegliere dove sta la centrale
		EventsDao dao = new EventsDao();
		Integer minD = dao.getDistrettoMin(anno);
		this.agenti.put(minD, this.N);

		//creo la coda
		this.queue = new PriorityQueue<Evento>();

		for(Event e : dao.listAllEventsByDate(this.anno, this.mese, this.giorno)) {
			queue.add(new Evento(TipoEvento.CRIMINE, e.getReported_date(), e));
		}
```

```
	}

	public int run() {
		Evento e;
		while((e = queue.poll()) != null) {
			switch(e.getTipo()) {

				case CRIMINE:
					System.out.println("NUOVO CRIMINE! " +
e.getCrimine().getIncident_id());
					Integer partenza = null;
					partenza = cercaAgente(e.getCrimine().getDistrict_id());

					if(partenza != null) {
						//c'Ã¨ un agente libero

	if(partenza.equals(e.getCrimine().getDistrict_id())) {
							//tempo di arrivo = 0
							System.out.println("AGENTE ARRIVA PER
CRIMINE: " + e.getCrimine().getIncident_id());
							Long duration =
getDuration(e.getCrimine().getOffense_category_id());
							this.queue.add(new
Evento(TipoEvento.GESTITO,

	e.getData().plusSeconds(duration),e.getCrimine()));

						} else {
							Double distance =
this.grafo.getEdgeWeight(this.grafo.getEdge(partenza,

	e.getCrimine().getDistrict_id()));
							Long seconds = (long) ((distance *
1000)/(60/3.6));
							this.queue.add(new
Evento(TipoEvento.ARRIVA_AGENTE,

	e.getData().plusSeconds(seconds), e.getCrimine()));
						}
					}else{
						//nessuno libero
						System.out.println("CRIMINE " +
e.getCrimine().getIncident_id() + " MAL GESTITO!!!!");
						this.malGestiti++;
					}
					break;

				case ARRIVA_AGENTE:
					System.out.println("AGENTE ARRIVA PER CRIMINE: " +
e.getCrimine().getIncident_id());
					Long duration =
getDuration(e.getCrimine().getOffense_category_id());
					this.queue.add(new Evento(TipoEvento.GESTITO,

	e.getData().plusSeconds(duration),e.getCrimine()));

					//controllo se il crimine Ã¨ mal gestito
```

```
		if(e.getData().isAfter(e.getCrimine().getReported_date().plusMinutes(15))) {
                                        System.out.println("CRIMINE " +
e.getCrimine().getIncident_id() + " MAL GESTITO!!!!");
                                        this.malGestiti++;
                                }
                                break;

                        case GESTITO:
                                System.out.println("CRIMINE " +
e.getCrimine().getIncident_id() + " GESTITO!");
                                this.agenti.put(e.getCrimine().getDistrict_id(),

		this.agenti.get(e.getCrimine().getDistrict_id()) +1);
                                break;
                        }
                }

                System.out.println("TERMINATO!! MAL GESTITI = " + this.malGestiti);
                return this.malGestiti;
        }

        private Integer cercaAgente(Integer district_id) {
                Double distanza = Double.MAX_VALUE;
                Integer distretto = null;

                for(Integer d : this.agenti.keySet()) {

                        if(this.agenti.get(d) > 0) {
                                if(district_id.equals(d)) {
                                        distanza = Double.valueOf(0);
                                        distretto = d;
                                }
                                else if(this.grafo.getEdgeWeight(this.grafo.getEdge(district_id,
d)) < distanza) {
                                        distanza =
this.grafo.getEdgeWeight(this.grafo.getEdge(district_id, d));
                                        distretto = d;
                                }
                        }
                }
                return distretto;
        }

        private Long getDuration(String offense_category_id) {
                if(offense_category_id.equals("all_other_crimes")) {
                        Random r = new Random();
                        if(r.nextDouble() > 0.5)
                                return Long.valueOf(2*60*60);
                        else
                                return Long.valueOf(1*60*60);
                } else
                        return Long.valueOf(2*60*60);
        }
}
```

- MODEL

```
public int simula(Integer anno, Integer mese, Integer giorno, Integer N) {
		Simulatore sim = new Simulatore();
		sim.init(N, anno, mese, giorno, grafo);
		return sim.run();
	}
```

SIMULAZIONE RICORSIVA

```
package it.polito.tdp.flightdelays.model;

import java.util.HashMap;
import java.util.List;
import java.util.Map;

import it.polito.tdp.flightdelays.db.FlightDelaysDAO;

public class Simulatore {

	//parametri
	private int passeggeri;
	private int voli;
	private FlightDelaysDAO dao;

	List<Flight> listaVoli;
	Map<Integer,Airport> aeroporti;

	//modello del mondo
	Map<Integer, Integer> passeggeroRitardo;

	//coda degli eventi e valori in output
	//private PriorityQueue<Integer> coda;

	public Simulatore(int passeggeri, int voli) {
		this.passeggeri = passeggeri;
		this.voli = voli;
		int i=0;
		passeggeroRitardo = new HashMap<>();
		dao = new FlightDelaysDAO();
		aeroporti = dao.loadAllAirports();
		while(i<passeggeri) {
			passeggeroRitardo.put(i, 0);
			i++;
		}
	}

	public int getPasseggeri() {
		return passeggeri;
	}

	public void setPasseggeri(int passeggeri) {
```

```
		this.passeggeri = passeggeri;
	}

	public int getVoli() {
		return voli;
	}

	public void setVoli(int voli) {
		this.voli = voli;
	}

	public void run() {
		for(int i=0; i<passeggeri; i++) {
			Flight primoVolo = dao.loadFlightsFromAirport(aeroporti.get(random()));
			int ritardo = primoVolo.getArrivalDelay()
+primoVolo.getDepartureDelay();
			int numVoli = voli-1;
			System.out.println("Primo volo: "+primoVolo.getOriginAirportId()+"
"+primoVolo.getDestinationAirportId()+"\n"+ritardo);
			int livello=1;
			continuaAVolare(ritardo,primoVolo,i,numVoli,livello);
		}
	}

	private int random() {
		double prob = Math.random();
		return (int) (prob*(aeroporti.size()-1));
	}

	private void continuaAVolare(int ritardo, Flight primoVolo, int passeggero, int
numVoli, int livello) {
		if(numVoli==0) {
			passeggeroRitardo.remove(passeggero);
			passeggeroRitardo.put(passeggero, ritardo);
			return;
		}
		if(dao.getProxVolo(primoVolo).getFlightNumber()!=0 && numVoli>0) {
			Flight proxVolo = dao.getProxVolo(primoVolo);
			primoVolo = proxVolo;
			ritardo+=(proxVolo.getArrivalDelay()+proxVolo.getDepartureDelay());
			numVoli--;
			//System.out.println("Prox Volo"+proxVolo.toString()+"\n"+ritardo);
			passeggeroRitardo.remove(passeggero);
			passeggeroRitardo.put(passeggero, ritardo);
			livello++;
			System.out.println("Volo n°"+livello+" :
"+proxVolo.getOriginAirportId()+"-"+proxVolo.getDestinationAirportId()+"\n"+ritardo);
			continuaAVolare(ritardo,proxVolo,passeggero,numVoli,livello);
		}

	}

	public Map<Integer, Integer> getPasseggeroRitardo() {
		return passeggeroRitardo;
	}


}
```

QUERY DATE DIFF

```
"SELECT COUNT(DISTINCT s1.id) AS c1, COUNT(DISTINCT s2.id) AS c2 \n"+
                          "FROM sighting s1, sighting s2 \n"+
                          "WHERE YEAR(s1.datetime) = ? AND YEAR(s1.datetime) =
YEAR(s2.datetime) \n"+
                          "AND s1.state = ? AND s2.state = ? AND
( (DATEDIFF(s1.datetime,s2.datetime) <= ? \n" +
                          "AND DATEDIFF(s1.datetime,s2.datetime) > 0 ) OR
(DATEDIFF(s2.datetime,s1.datetime) <= ? \n" +
                          "AND DATEDIFF(s2.datetime,s1.datetime) > 0) )"
```