University of Aveiro
Computer and Telematics Engineering
Data and Knowledge Engineering

Airlines.

André Pinho
Nº mec: 80313

Ana Margarida Silva
Nº mec: 77752

Diego Hernandez
Nº mec: 77013

21th of December of 2018
Aveiro

universidade
de aveiro

Index

1. Introduction

AirlinesDot is a web based application capable of finding flight routes given a source and a destination point. The routes are visualized on a interactable map. AirlinesDot is able to show more than one possible flight route thanks to a search algorithm capable of calculating a route with the given parameters resulting the best ones in terms of cost, distances, number of hops and other factors. The client can locate points of interest of a city and search the location of an exact point of interest, which for example could be a street, museum or any kind of tourist attraction. Finally, this application has also a smart search feature that allows the user to input two locations, not being necessarily an airport, and get the best territorial route and flight route between the departure and arrival location.

This application offers an API, which allows other applications to do read-only operations in our system.

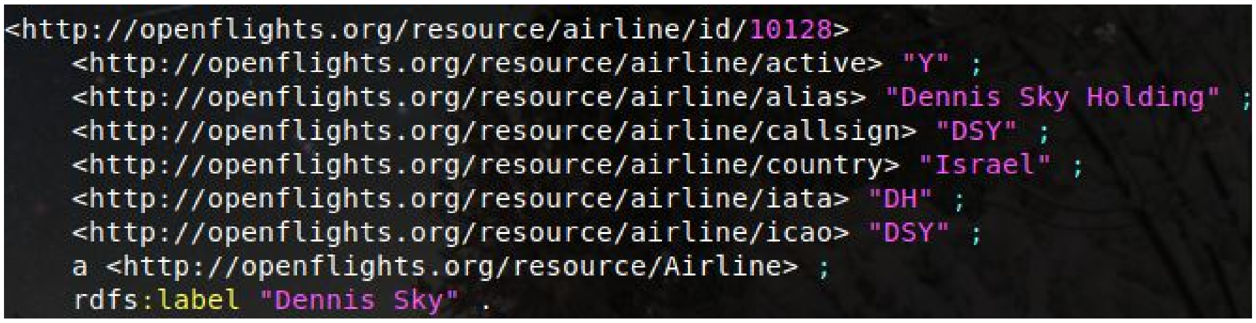
For the development of this website we used Django Framework to create the pages and URLs with the addition of javascript to handle the map. Data used to process the gathered data about flights, airports and airlines was stored in our own triplestore (GraphDB). To query the data in the triplestore, we used SPARQL. We also used RDFa on our website which allows crawlers and intelligent clients to extract meaningful information from our service. Our app also uses Wikidata as a way to get more data such as monuments, points of interest of a city and to allow our service to have an elastic and flexible way to get the location of arbitrary points of interest.

Our mindset for this project was to make of use all the advantages of the technology mentioned above and unite algorithms and concepts learnt from other areas of Information and Technology, such as graphs, cache, and others.

2. Data Source and Transformation

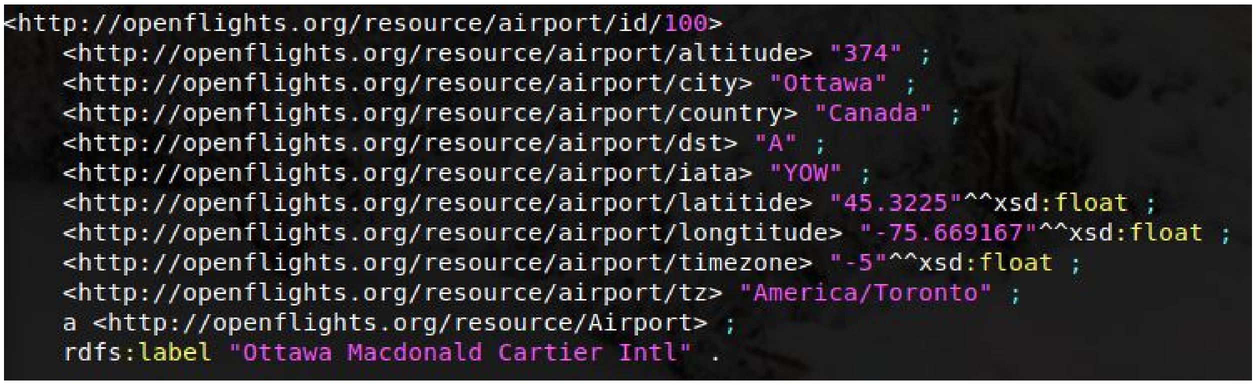
2.1 Flight data

The main data source of our project is a dataset supplied by the GraphDB documentation (http://graphdb.ontotext.com/free/_downloads/airroutes.ttl.zip). This dataset is used as an example to "Visualize GraphDB data with Ogma JS" (http://graphdb.ontotext.com/free/devhub/map.html).
This dataset contains about 224000 triples and uses the information given by the OpenFlights dataset (https://openflights.org/data.html). It's located in our project on the 'datasets/' folder. As given, this dataset contains data about Airlines, Airports and Flight Routes:



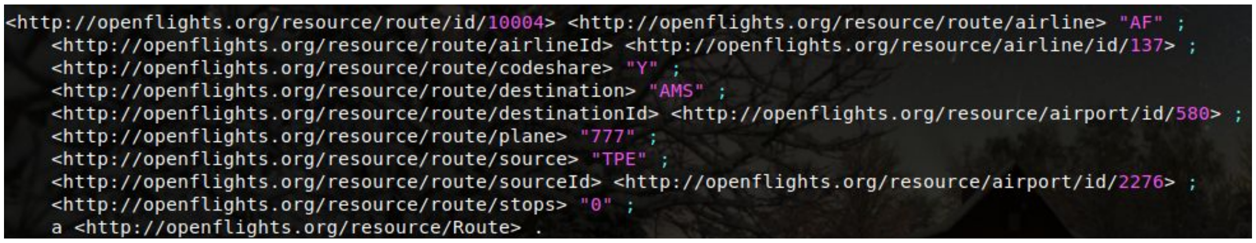
```
<http://openflights.org/resource/airline/id/10128>
    <http://openflights.org/resource/airline/active> "Y" ;
    <http://openflights.org/resource/airline/alias> "Dennis Sky Holding" ;
    <http://openflights.org/resource/airline/callsign> "DSY" ;
    <http://openflights.org/resource/airline/country> "Israel" ;
    <http://openflights.org/resource/airline/iata> "DH" ;
    <http://openflights.org/resource/airline/icao> "DSY" ;
    a <http://openflights.org/resource/Airline> ;
    rdfs:label "Dennis Sky" .
```

Example of an Airline.



```
<http://openflights.org/resource/airport/id/100>
    <http://openflights.org/resource/airport/altitude> "374" ;
    <http://openflights.org/resource/airport/city> "Ottawa" ;
    <http://openflights.org/resource/airport/country> "Canada" ;
    <http://openflights.org/resource/airport/dst> "A" ;
    <http://openflights.org/resource/airport/iata> "YOW" ;
    <http://openflights.org/resource/airport/latitide> "45.3225"^^xsd:float ;
    <http://openflights.org/resource/airport/longtitude> "-75.669167"^^xsd:float ;
    <http://openflights.org/resource/airport/timezone> "-5"^^xsd:float ;
    <http://openflights.org/resource/airport/tz> "America/Toronto" ;
    a <http://openflights.org/resource/Airport> ;
    rdfs:label "Ottawa Macdonald Cartier Intl" .
```

Example of an Airport.



```
<http://openflights.org/resource/route/id/10004> <http://openflights.org/resource/route/airline> "AF" ;
    <http://openflights.org/resource/route/airlineId> <http://openflights.org/resource/airline/id/137> ;
    <http://openflights.org/resource/route/codeshare> "Y" ;
    <http://openflights.org/resource/route/destination> "AMS" ;
    <http://openflights.org/resource/route/destinationId> <http://openflights.org/resource/airport/id/580> ;
    <http://openflights.org/resource/route/plane> "777" ;
    <http://openflights.org/resource/route/source> "TPE" ;
    <http://openflights.org/resource/route/sourceId> <http://openflights.org/resource/airport/id/2276> ;
    <http://openflights.org/resource/route/stops> "0" ;
    a <http://openflights.org/resource/Route> .
```

Example of a Flight Route.

Although the dataset seems complete, there are a few critical attributes that are essential to our application. These are, for example, cost of a flight, distance of a flight, time of arrival, day of arrival, time of departure and day of departure.
Since we couldn't find this data (even the openflights website which had the same data as this dataset had, but with no timetables), we decided to generate the remaining parameters with a python script. This script, located at 'datasets/converter.py', is responsible for generating random 'close-to-real' values for the missing attributes. It is also responsible for filtering incomplete data (some airports don't seem to have source or destination URIs, and some routes refer to a non existent airport) and fixing a syntax typo on the coordinates. It uses RDFLib to load the original dataset, manipulate it (insert and remove data) and store it on new file. It also uses SPARQL as means of retrieving info from the original dataset (described in more detail on the section 3.1).
First, the script starts by generating two temporary random attributes to the airlines, the base cost of a flight and the cost per hundred kilometers (function 'getairlinesdata'). Then, for each flight, the distance between the two airports is calculated, using the Haversine function. This distance and the newly calculated parameters of the route's airline are used to give a different cost to each flight, even if it is between the same airports and done by different airlines. Since the original dataset doesn't contain time info, we assumed that the flights are repeated every day at the same hours. So, after generating a random start time to each flight, we generate a random velocity between 740Km/h and 930Km/h (average commercial airplane velocities) and use it to calculate the duration of the flight. The time of arrival is also calculated using these parameters. All of this route parameters are generated using the 'getroutesdata' function.
Finally, after calculating all the new info, it stores it on the file ('airlinesdot.n3'), using format N3, with 'http://www.airlinesdot.com/resource/route/' as the base URI for the new triples. After this conversion, the dataset ends up with approximately 277000 triples.
We already provide the final RDF N3 dataset, but it can be re-generated calling the following command:

```
$ python converter.py
```

The final file is used as one of our datasets in a readonly database called 'airlinesdot'.

Although the dataset seems extensive, it’s still not a complete set. This means that two random airports have a significant chance of not being connected. Adding to that, there are some heliports disguised as airports in our dataset (ex: Wall Street Heliport, located in New York). Through this project report we will enumerate some examples that work well with our system.
Another thing to note is that since the hours are random, the waiting time between flights becomes unrealistic, since in real life airlines schedule flights to allow for coherent scales.
Finally, the cost of the flights tend to skew over the real ones, again due to our model guessing the cost using the distance of the flight and some randomness.

3. Data operations (using SPARQL)

SPARQL is a RDF query language able to retrieve and manipulate data stored in a triplestore. Most operations on the data used for this web application comes from SPARQL technology.
For a better comprehension we will talk about its operations according to the functionality, web application or file integrated on the website:

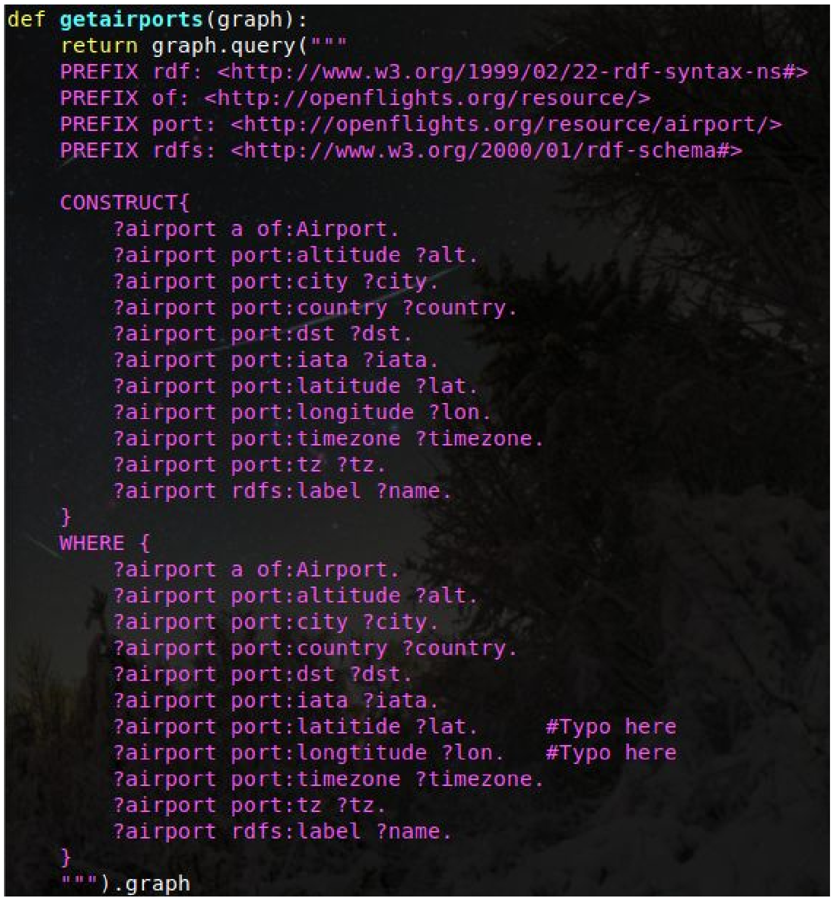
3.1 Dataset Converter

Dataset Converter ('datasets/converter.py') is a python script which uses SPARQL technology to operate on unprocessed data to generate our final dataset, as mentioned above on chapter 2.1. This file also uses the python library RDFLib to work with RDF, a simple yet powerful language for representing information on graphs.
Through SPARQL this script retrieves data about the existing airlines through the function 'getairlines'. It uses the CONSTRUCT search to get the complete triples from the 'datasets/airrotes.ttl' data. Predicates in order:

- Type of the subject (in this case Airline)
- The callsign of the airline;
- The Country of the airline;
- The IATA (*International Air Transport Association*) of the Airline;
- The ICAO (*International Civil Aviation Organization*) of the Airline;
- The name of the Airline.

This script also retrieves data about relevant information of the available airports on the database file using the 'getairports' function. Also, this function corrects a typo made on the original dataset: it's written "latitide" where it should be "latitude", and "longtitude" where it should be "longitude". So, this query uses CONSTRUCT not only to return full triples, but also to fix that issue. Predicates in order:

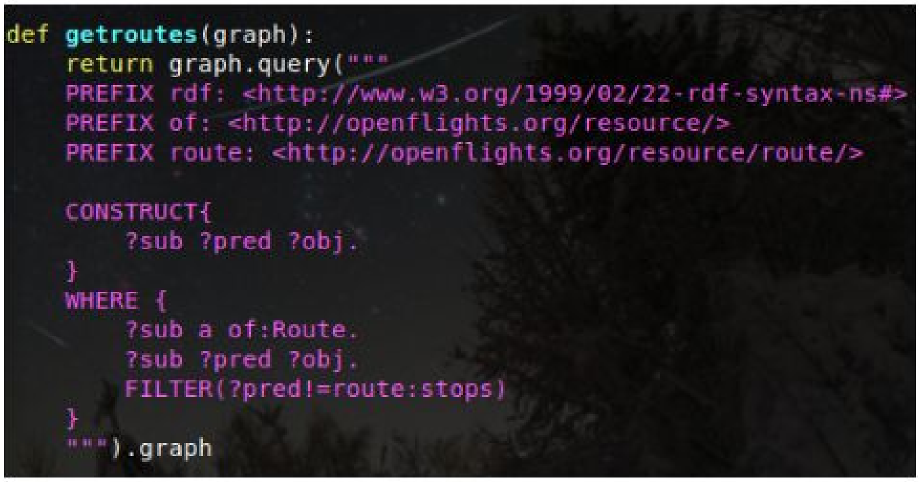
- The type of the subject (for this case Airport);
- The Altitude of the Airport;
- The city of the Airport;
- The country of the Airport;
- The Daylight saving time;
- The IATA of the Airport;
- The latitude of the Airport;
- The longitude of the Airport;
- The time zone where the the Airport belongs;
- And the name/label of the Airport.



```
def getairports(graph):
    return graph.query("""
    PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
    PREFIX of: <http://openflights.org/resource/>
    PREFIX port: <http://openflights.org/resource/airport/>
    PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

    CONSTRUCT{
        ?airport a of:Airport.
        ?airport port:altitude ?alt.
        ?airport port:city ?city.
        ?airport port:country ?country.
        ?airport port:dst ?dst.
        ?airport port:iata ?iata.
        ?airport port:latitude ?lat.
        ?airport port:longitude ?lon.
        ?airport port:timezone ?timezone.
        ?airport port:tz ?tz.
        ?airport rdfs:label ?name.
    }
    WHERE {
        ?airport a of:Airport.
        ?airport port:altitude ?alt.
        ?airport port:city ?city.
        ?airport port:country ?country.
        ?airport port:dst ?dst.
        ?airport port:iata ?iata.
        ?airport port:latitide ?lat.     #Typo here
        ?airport port:longtitude ?lon.   #Typo here
        ?airport port:timezone ?timezone.
        ?airport port:tz ?tz.
        ?airport rdfs:label ?name.
    }
    """).graph
```

'getairports' function

Finally, this module also has a function to get routes ('getroutes' function) from the original dataset. Again, it uses the CONSTRUCT and it returns all the triples except the ones that specifiy the number of stops (redundant information, they were always 1):



```
def getroutes(graph):
    return graph.query("""
    PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
    PREFIX of: <http://openflights.org/resource/>
    PREFIX route: <http://openflights.org/resource/route/>

    CONSTRUCT{
        ?sub ?pred ?obj.
    }
    WHERE {
        ?sub a of:Route.
        ?sub ?pred ?obj.
        FILTER(?pred!=route:stops)
    }
    """).graph
```

'getroutes' function

3.2 Query Collection

This module, located at ‘datasets/querycollection.py’, provides functions that return SPARQL queries targeted at our readonly database (‘airlinesdot’) to various of our services:

getRoutesAirport:
Given the URI of an Airport stored on our database, we retrieve data about the all the destinations that gets place on the correspondent airport. We also retrieve data about the distance of the route, its price, duration, time of arrival and the latitude and longitude of the destination airport.

getAirportCoords:
This self-explicit function get the coordinates, latitude and longitude of an Airport given its URI.

getAirportCity:
Given the name of a city, this function retrieves all airports of it. The returned values are the airport’s URI, the values of its longitude and latitude, the label of the airport, its IATA, country and city where the airport belongs.

getAirportURI:
Given the name of the Airport, this module returns its URI and, additionally, its latitude and longitude.

getAIrportCoord:
Given the URI of the airport, this module retrieves the latitude and longitude of the correspondent Airport.

getInfoRoute:
Given the URI of a specific route, this module retrieves needed data about it, such as: airline, destination, the airplane, point of departure and arrival. Additionally, it also returns its URI, the distance and duration of the flight and ,finally , the time of arrival and departure.

getCitysWithAirports:
Given the name of a country it returns all the names of the cities, registered on the database, that have at least one airport.

getAirportsFromCountry:
Given the literal of a Country, it returns the URI of the Airport that belongs to that country and the coordinates (latitude and longitude) of the airport.

getAirportInfo:
Given the URI of an Airport registered on the database, it retrieves the coordinates of the Airport (latitude and longitude), its IATA, Country and City where it belongs and finally the label of the Airport.

3.4 Shortest Path Search

To be able to get the best routes available between two arbitrary places that might not be directly connected, we decided to develop a search algorithm (‘datasets/searchRoutes.py’). We also decided that this algorithm should optimize the path for different parameters. These are price of the whole trip, distance traveled, number of times a passenger would change airplane (hops), total trip time (including waiting in the airport), and flight time only.
For this application, we figured that an A* (A star) search algorithm would be a good option. For it, we needed a cost function and a heuristic function.
The cost function gives the total cumulative cost of a jump between two connected nodes. The cost function varies for each optimization parameter:

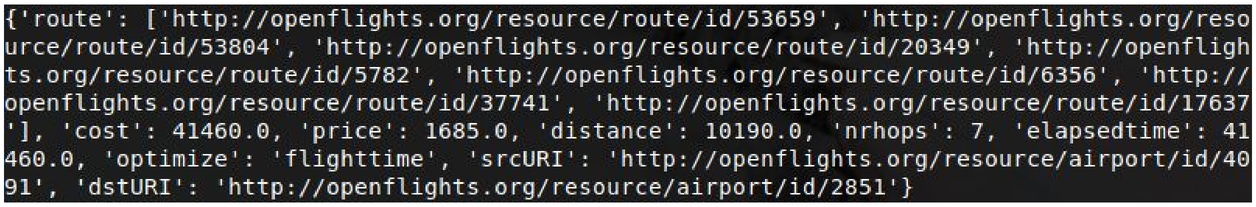
- Price: The cost is equal to the sum of the node’s cumulative cost and the new jump’s price;
- Distance: The cost is equal to the sum of the node’s cumulative cost and the new jump’s length;
- Hop: The cost is equal to the increment of the last cost (each new jump, the total cost increases by 1);
- Time: This one is not so trivial to implement. First, the initial jump’s cost is equal to the sum of the duration of the first flight with the hour. For the next jumps:
  - If it’s possible to catch the flight today, then the cost is equal to the sum of the accumulative time, waiting time and the duration of the flight;
  - Else, the cost is equal to the sum of the cumulative time, the time needed to reach midnight, the time of the flight and the duration of the flight;

  To implement all these cases, we created auxiliary functions to convert python’s Time, Date and DateTimes to seconds, to allow for a more flexible manipulation. This means that the cost can still be a number and not a structure;
- Flight time: same as Time, but the waiting times are equal to zero, so only the actual time flying is accounted.

The heuristic function gives the estimated cost from a certain node to the final destination. The heuristic function varies for each optimization parameter:

- Price: Assuming the cheapest possible price if the rest of the flight is a direct connection (cost calculated using the cheapest possible parameters);
- Distance: Shortest possible distance between the node and the destination;
- Hop, Time and FlightTime : No trivial way to predict, so it is always zero.

While experimenting with the algorithm, we found out that we would get slightly better results on not that much time if we ignored the heuristic function. So, we decided to keep the function, but override it with a ‘return 0’. This means the search is now a particular case of a A* search: it’s a uniform cost search, also known as the Dijkstra’s (Shortest Path) Algorithm.
For each iteration of the algorithm, the cheapest possible alternative is processed. Using SPARQL, it is retrieved from our database the possible jumps (function ‘possible_hops’) including all the information needed to calculate the costs (URI of the new airport, coordinates of the new airport, route URI, price, distance, duration and start time of the route). This query uses the ‘getRoutesAirport’ function stored at the query collection code (explained on section 3.2).
To get the destination and source airports coordinates, the configurator (function ‘configure’ of the class routeFinder) queries our database with the query present on the function ‘getAirportCoords’.
This module offers two interfaces to the caller, the routeFinderURI and the routeFinderNAME.
The first one receives the source and destination airport URIs, calculates the best routes (using the discussed techniques) and returns the path found, or None, if such path doesn’t exist. The latter one receives the source and destination city names, finds their airports (URI) through the SPARQL and the getAirportCity functions, generates all possible combinations of source and destination airports and calls routeFinderURI until a valid result is found.
In both cases, the output of the of the module is a dictionary that includes all the relevant results.


```
{'route': ['http://openflights.org/resource/route/id/53659', 'http://openflights.org/reso
urce/route/id/53804', 'http://openflights.org/resource/route/id/20349', 'http://openfligh
ts.org/resource/route/id/5782', 'http://openflights.org/resource/route/id/6356', 'http://
openflights.org/resource/route/id/37741', 'http://openflights.org/resource/route/id/17637
'], 'cost': 41460.0, 'price': 1685.0, 'distance': 10190.0, 'nrhops': 7, 'elapsedtime': 41
460.0, 'optimize': 'flighttime', 'srcURI': 'http://openflights.org/resource/airport/id/40
91', 'dstURI': 'http://openflights.org/resource/airport/id/2851'}
```

Extract of the output dictionary (optimizing the flight time)

As a way to minimize the time needed to calculate routes, we implemented a N-thread pool to parallelize the different instances of the algorithm. To our surprise, this modification did not improve the times by a significant amount (average less 9 seconds). After analysing the code and all the call trace, we concluded that it is due to graphDB representing a bottleneck in the algorithm: at each iteration of the algorithm, the database is queried for the possible connections of each node. Parallelizing the algorithm would only bring improvements by acting in the time between queries, and for this case, it is almost insignificant. Having concluded this, we decided to remove the code since it was increasing the complexity of an already somewhat complex module.

3.5 Cache Manager

As another way to mitigate the time spent searching for solutions, we implemented a cache system using the SPARQL update functionality of GraphDB.
The main objective of the cache is to store the results of already searched flight routes. This way, the overall processing time can be reduced since searching a solution already in cache is significantly faster than recalculating the route.

The Cache Manager ('datasets/searchRoutes.py') makes of use of another GraphDB repository named 'airlinesdotCache' where triples about flights and routes are stored, consulted and deleted. This database is separated from the main GraphDB ('airlinesdot') for better performance: the main database is very big (almost 300.000 triples), so merging both databases would deteriorate the performance of queries.

Each flight stored on this cache is composed of a linked list in which the first element identifies the complete route, costs, and other parameters, while the remaining items of the linked list point (in order) to the routes used.

The two main functions of this module are the 'addtocache' and the 'retrivefromcache'.

The function 'addtocache' receives a dictionary that contains a route , checks if its not already in cache, and stores it. This function is called for each optimization variable. The input dictionary is the same as the output dictionary of the shortest path module, shown in the last section (3.4)

So, for each route, an unique URI that identifies the complete flight (function get_next_pflight) is created. With that URI, a structure similar to a header (first node of the linked list) is created, where the information such as the source airport URI, destination URI, the optimization variable, total cost, total price, total distance, total number of jumps and elapsed times is stored. It also stores a parameter that identifies the validity of the route, which is 1 month after the date of registration. This means that after a month of caching that route, the route is discarded and recalculated.
With that, the other nodes are generated, also with an unique URI (function 'get_next_proute'). Each node contains a pointer to the actual route, and (if it exists) the next linked list node. The last node doesn't have a 'next' predicate.

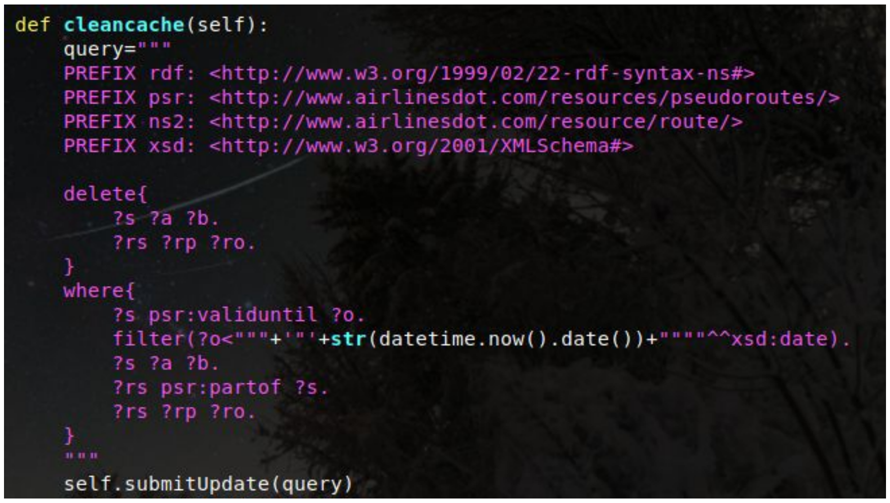
The following data is an extract of the cache:

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix ns2: <http://www.airlinesdot.com/resource/route/> .
@prefix psr: <http://www.airlinesdot.com/resources/pseudoroutes/> .

psr:flight_0 a psr:PseudoFlight .
psr:flight_0 psr:flightid 0 .
psr:flight_0 psr:validuntil "2019-01-20"^^xsd:date .
Psr:flight_0 ns2:sourceId <http://openflights.org/resource/airport/id/4091> .
psr:flight_0 ns2:destinationId
<http://openflights.org/resource/airport/id/2851> .
psr:flight_0 psr:optimize "price" .
psr:flight_0 psr:cost 1558.0 .
psr:flight_0 psr:price 1558.0 .
psr:flight_0 psr:distance 12512.0 .
psr:flight_0 psr:nrhops 5 .
psr:flight_0 psr:elapsedtime 277873.0 .
psr:flight_0 psr:next psr:route_0 .

psr:route_0 a psr:PseudoRoute .
psr:route_0 psr:routeid 0 .
psr:route_0 psr:next psr:route_1 .
psr:route_0 psr:partof psr:flight_0 .
psr:route_0 psr:route <http://openflights.org/resource/route/id/48376> .

psr:route_1 a psr:PseudoRoute .
psr:route_1 psr:next psr:route_2 .
psr:route_1 psr:partof psr:flight_0 .
psr:route_1 psr:route <http://openflights.org/resource/route/id/48400> .

psr:route_2 a psr:PseudoRoute .
psr:route_2 psr:next psr:route_3 .
psr:route_2 psr:partof psr:flight_0 .
psr:route_2 psr:route <http://openflights.org/resource/route/id/32695> .

psr:route_3 a psr:PseudoRoute .
psr:route_3 psr:next psr:route_4 .
psr:route_3 psr:partof psr:flight_0 .
psr:route_3 psr:route <http://openflights.org/resource/route/id/12418> .

psr:route_4 a psr:PseudoRoute .
psr:route_4 psr:partof psr:flight_0 .
psr:route_4 psr:route <http://openflights.org/resource/route/id/17579> .
```

The function ‘retrivefromcache’ receives a source airport URI and a destination airport URI. First, this function calls the ‘cleancache’ function (which erases expired entries in the cache). Then, with boths URIs, it checks if there is an entry on the cache that already has those two parameters. If there is, it returns it, otherwise, it returns None.
To retrieve from the cache, it starts by finding the first node of the linked list (the header) through a query.

After finding the header (and extracting from it all the information), it starts traversing the linked list, storing each route in a python list until it reaches the final node. After doing this for all the optimize variables (price, distance, number of hops, total time and flight time), it generates a dictionary with the same structure as the one accepted by the ‘addtocache’ function, making the existence of the cache practically invisible to the caller.

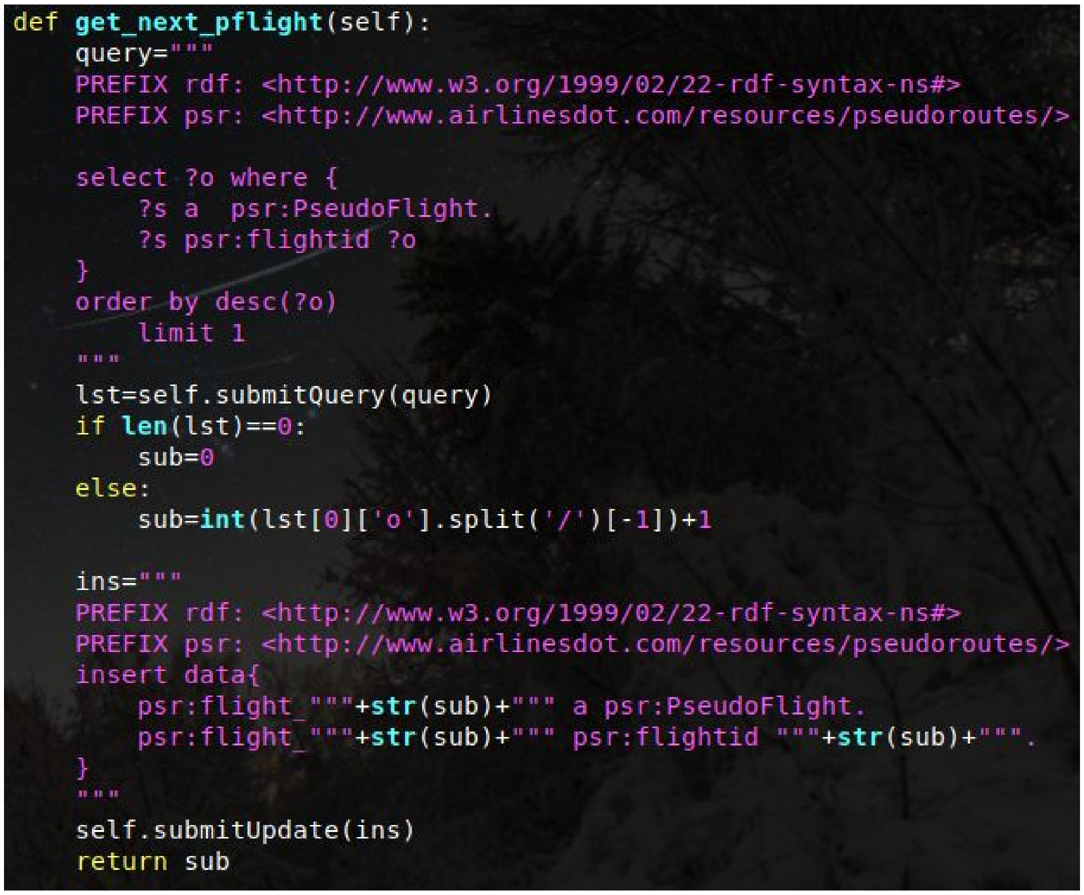
To clear the cache, the function ‘cleancache’ uses a SPAQRL delete, where the object of the predicate ‘validuntil’ is filtered)



```
def cleancache(self):
    query="""
    PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
    PREFIX psr: <http://www.airlinesdot.com/resources/pseudoroutes/>
    PREFIX ns2: <http://www.airlinesdot.com/resource/route/>
    PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

    delete{
        ?s ?a ?b.
        ?rs ?rp ?ro.
    }
    where{
        ?s psr:validuntil ?o.
        filter(?o<"""+'"'+str(datetime.now().date())+""""^^xsd:date).
        ?s ?a ?b.
        ?rs psr:partof ?s.
        ?rs ?rp ?ro.
    }
    """
    self.submitUpdate(query)
```

‘cleancache’ function

Both functions ‘get_next_pflight’ and ‘get_next_proute’ work in a similar way. Both query the ‘airlinesCache’ database to get most recent URI, use it to generate a new one (incrementing the ID by one) and store a ‘type’ triple as a way of claiming that URI:



```
def get_next_pflight(self):
    query="""
    PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
    PREFIX psr: <http://www.airlinesdot.com/resources/pseudoroutes/>

    select ?o where {
        ?s a  psr:PseudoFlight.
        ?s psr:flightid ?o
    }
    order by desc(?o)
        limit 1
    """
    lst=self.submitQuery(query)
    if len(lst)==0:
        sub=0
    else:
        sub=int(lst[0]['o'].split('/')[-1])+1

    ins="""
    PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
    PREFIX psr: <http://www.airlinesdot.com/resources/pseudoroutes/>
    insert data{
        psr:flight_"""+str(sub)+""" a psr:PseudoFlight.
        psr:flight_"""+str(sub)+""" psr:flightid """+str(sub)+""".
    }
    """
    self.submitUpdate(ins)
    return sub
```

‘get_next_pflight’ function

4. Data publication

4.1 Web API

To better modularize and isolate our services, we created an API that exposes some of the data. This can be used internally to simplify data requests from the webpage but simultaneously expands the usefulness of our services, since external applications can also make use of it. These services can be found by accessing the respective url:

- **List airport info in given city:**
  - (localhost:8000)/api/city/***<city_name>***/airports (HTML + RDFa)

- **List monuments in given city:**
  - /api/city/***<city_name>***/monuments

- **Information about source and destination coordinates, as well as the information of each route flight*:***
  - /api/city/orig/***<origin_city>***/dest/***<destination_city>***/***<date>***/coord (json)

- **List airports in given country:**
  - /api/country/***<country_name>***/airports (array)

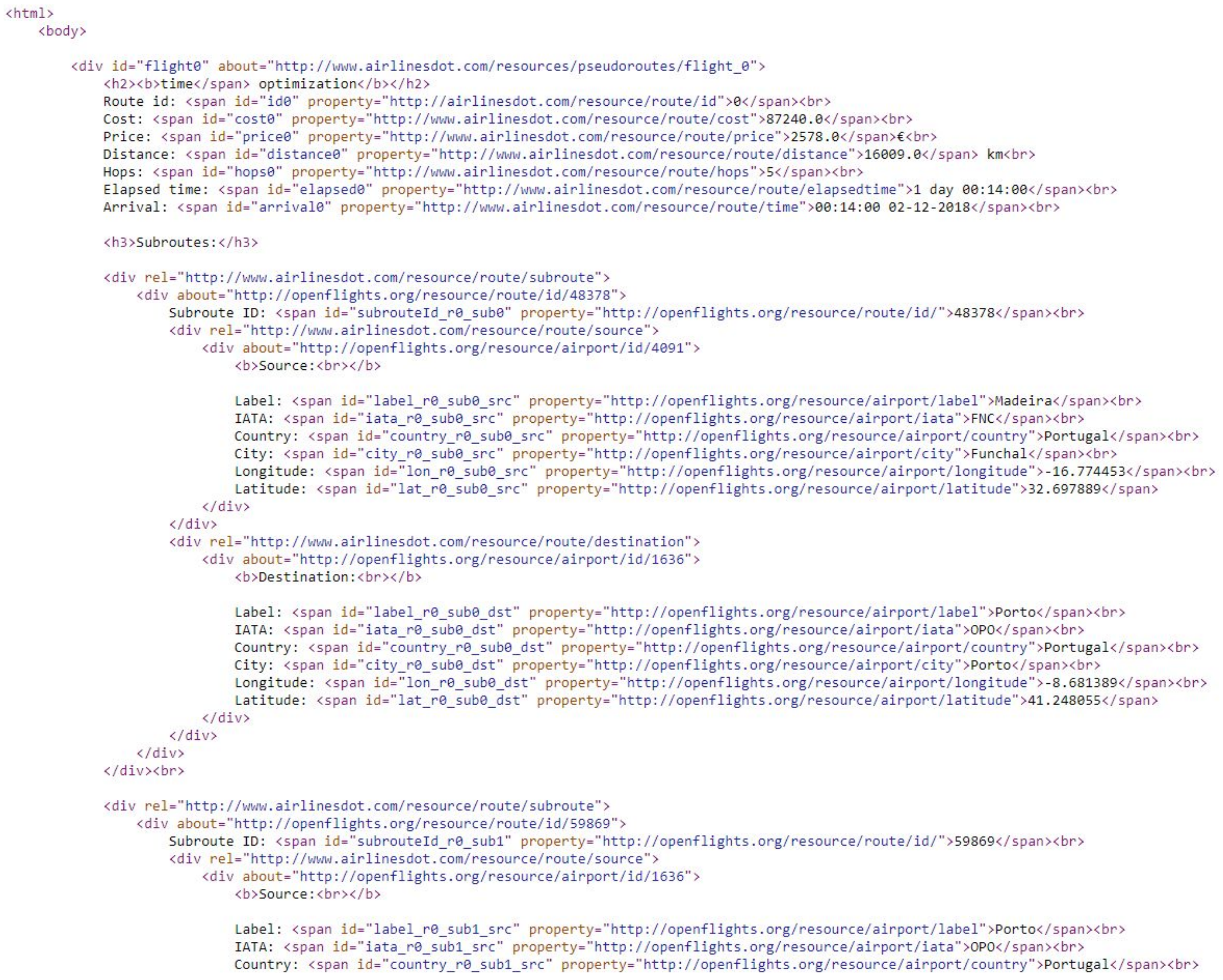
- **List cities with airports in given country:**
  - /api/country/***<country_name>***/cities (list)

- ***Information about the point of interest and its coordinates***
  - api/monument/**<point_of_interest>** (json)

- **Get subroute info betweem origin/destination cities:**
  - /api/routes/***<origin_city>***/***<dest_city>***/***<year-month-day>*** (HTML + RDFa)

4.2 Publication of semantic data through RDFa

When using our application to search for a flight, besides being able to visualize the routes on the map, all related data will be displayed in the page as text. Flight information is obtained by requesting routes from a source to a destination to our Web API, which in turn returns the results already formatted in HTML with RDFa. This means that in addition to presenting the data to the user, such information also contains semantic meaning that can be understood by machines. Since it isn't required to interact directly with the webpage to have access to the data, as it can be retrieved from our API, external services can easily make use of it.
In order to see the RDFa, "View source" or "Inspect element" can be used either after searching for a flight on the first section of our website (more instructions on how to do that later on) or by sending a request to our API to either list airports in a city or to get routes between two cities. The example below contains part of the reply after requesting routes between Funchal and Caracas.

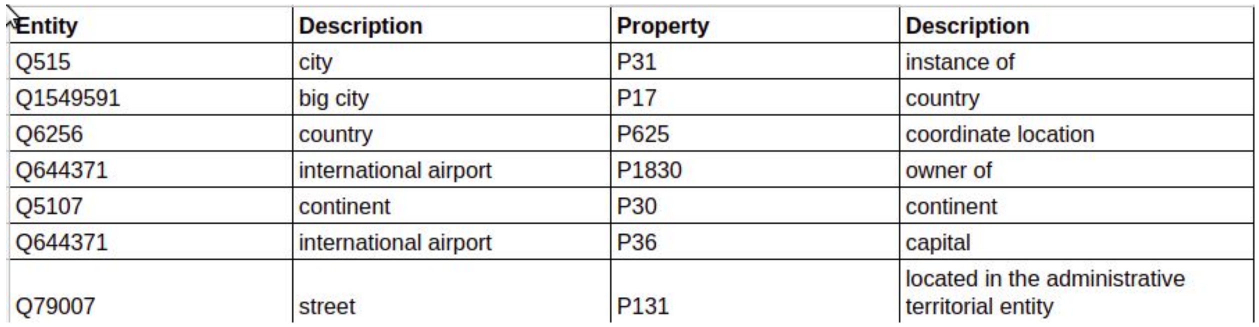


```
<html>
    <body>

        <div id="flight0" about="http://www.airlinesdot.com/resources/pseudoroutes/flight_0">
            <h2><b>time</span> optimization</b></h2>
            Route id: <span id="id0" property="http://airlinesdot.com/resource/route/id">0</span><br>
            Cost: <span id="cost0" property="http://www.airlinesdot.com/resource/route/cost">87240.0</span><br>
            Price: <span id="price0" property="http://www.airlinesdot.com/resource/route/price">2578.0</span>€<br>
            Distance: <span id="distance0" property="http://www.airlinesdot.com/resource/route/distance">16009.0</span> km<br>
            Hops: <span id="hops0" property="http://www.airlinesdot.com/resource/route/hops">5</span><br>
            Elapsed time: <span id="elapsed0" property="http://www.airlinesdot.com/resource/route/elapsedtime">1 day 00:14:00</span><br>
            Arrival: <span id="arrival0" property="http://www.airlinesdot.com/resource/route/time">00:14:00 02-12-2018</span><br>

            <h3>Subroutes:</h3>

            <div rel="http://www.airlinesdot.com/resource/route/subroute">
                <div about="http://openflights.org/resource/route/id/48378">
                    Subroute ID: <span id="subrouteId_r0_sub0" property="http://openflights.org/resource/route/id/">48378</span><br>
                    <div rel="http://www.airlinesdot.com/resource/route/source">
                        <div about="http://openflights.org/resource/airport/id/4091">
                            <b>Source:<br></b>

                            Label: <span id="label_r0_sub0_src" property="http://openflights.org/resource/airport/label">Madeira</span><br>
                            IATA: <span id="iata_r0_sub0_src" property="http://openflights.org/resource/airport/iata">FNC</span><br>
                            Country: <span id="country_r0_sub0_src" property="http://openflights.org/resource/airport/country">Portugal</span><br>
                            City: <span id="city_r0_sub0_src" property="http://openflights.org/resource/airport/city">Funchal</span><br>
                            Longitude: <span id="lon_r0_sub0_src" property="http://openflights.org/resource/airport/longitude">-16.774453</span><br>
                            Latitude: <span id="lat_r0_sub0_src" property="http://openflights.org/resource/airport/latitude">32.697889</span>
                        </div>
                    </div>
                    <div rel="http://www.airlinesdot.com/resource/route/destination">
                        <div about="http://openflights.org/resource/airport/id/1636">
                            <b>Destination:<br></b>

                            Label: <span id="label_r0_sub0_dst" property="http://openflights.org/resource/airport/label">Porto</span><br>
                            IATA: <span id="iata_r0_sub0_dst" property="http://openflights.org/resource/airport/iata">OPO</span><br>
                            Country: <span id="country_r0_sub0_dst" property="http://openflights.org/resource/airport/country">Portugal</span><br>
                            City: <span id="city_r0_sub0_dst" property="http://openflights.org/resource/airport/city">Porto</span><br>
                            Longitude: <span id="lon_r0_sub0_dst" property="http://openflights.org/resource/airport/longitude">-8.681389</span><br>
                            Latitude: <span id="lat_r0_sub0_dst" property="http://openflights.org/resource/airport/latitude">41.248055</span>
                        </div>
                    </div>
                </div>
            </div><br>

            <div rel="http://www.airlinesdot.com/resource/route/subroute">
                <div about="http://openflights.org/resource/route/id/59869">
                    Subroute ID: <span id="subrouteId_r0_sub1" property="http://openflights.org/resource/route/id/">59869</span><br>
                    <div rel="http://www.airlinesdot.com/resource/route/source">
                        <div about="http://openflights.org/resource/airport/id/1636">
                            <b>Source:<br></b>

                            Label: <span id="label_r0_sub1_src" property="http://openflights.org/resource/airport/label">Porto</span><br>
                            IATA: <span id="iata_r0_sub1_src" property="http://openflights.org/resource/airport/iata">OPO</span><br>
                            Country: <span id="country_r0_sub1_src" property="http://openflights.org/resource/airport/country">Portugal</span><br>
```

universidade
de aveiro

5. Wikidata data integration

Wikidata is a free open knowledge base that can be read and edited by humans and machines. Wikidata is focused on items (entities) that are identified by a unique id number, prefixed with the letter ‘Q’. The properties of a triplet describes the data value of a statement, which can also be qualifiers. Properties have their own pages on Wikidata and are connected to items, resulting in a linked data structure, where the property represents a triplet's predicate. In Wikidata, the properties are prefixed with the letter ‘P’.



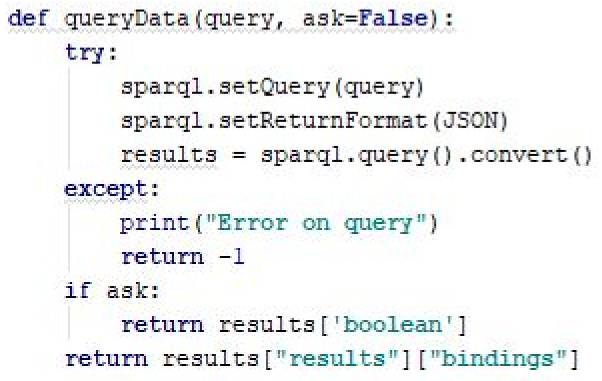
Entity	Description	Property	Description
Q515	city	P31	instance of
Q1549591	big city	P17	country
Q6256	country	P625	coordinate location
Q644371	international airport	P1830	owner of
Q5107	continent	P30	continent
Q644371	international airport	P36	capital
Q79007	street	P131	located in the administrative territorial entity

Some of the used Items and Properties IDs

‘datasets/querywikidata.py’ is one of the files that takes advantage of both SPARQL and Wikidata service.

The file has dependencies of the library SPARQLWrapper to make of use of Wikidata and JSON as a means to retrieve the data.
SPARQLWrapper is initialized giving the following url: https://query.wikidata.org/sparql.

queryData:
On this module we ask and retrieve data from Wikidata, by setting the query and receiving the response on JSON Format. This module returns the response.



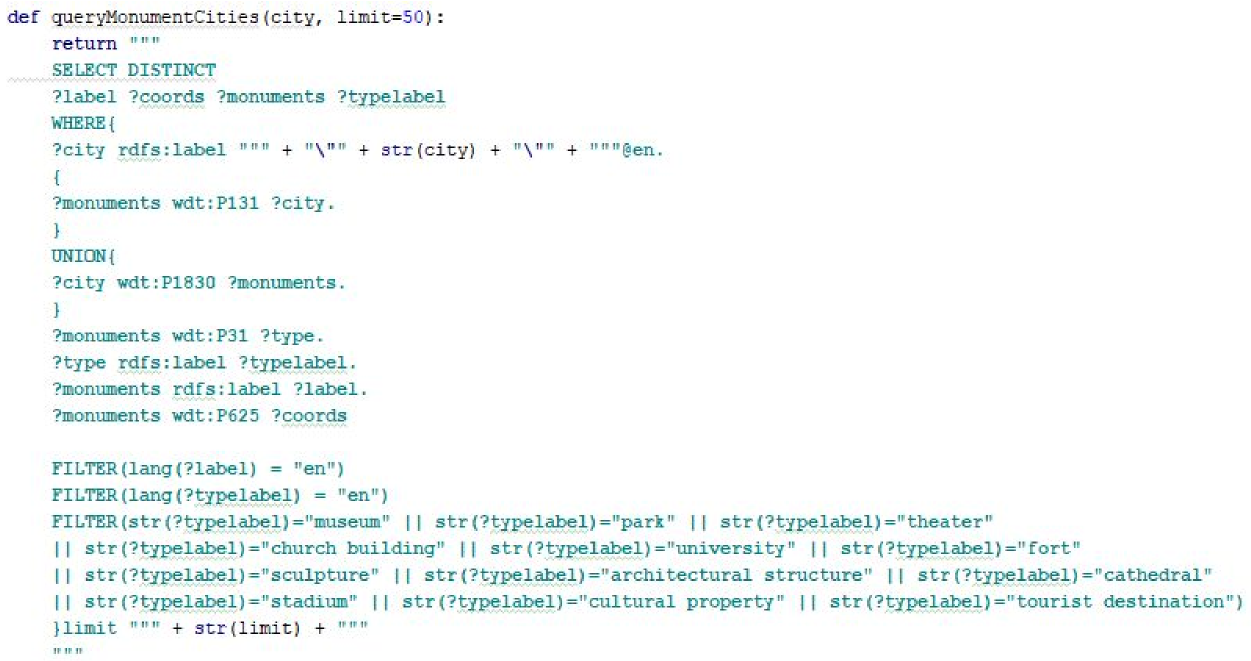
```
def queryData(query, ask=False):
    try:
        sparql.setQuery(query)
        sparql.setReturnFormat(JSON)
        results = sparql.query().convert()
    except:
        print("Error on query")
        return -1
    if ask:
        return results['boolean']
    return results["results"]["bindings"]
```

queryCountryAirports:
Given a Country Literal, the query represented on the correspondent module will retrieve the labels of the cities of a country and its coordinates, as long as it is an instance (P31) of a big city (Q1549591) or a city (Q515). The retrieved cities literals are independent of its coordinates of the URI that represents it, this is it is optional.
The labels are filtered with the goal of just retrieving the english labels of the cities of the country.

queryIsCity:
A query of type ASK, verifies the subject or URI of a given a literal if it is a is a type of (P31) city (Q515) or a big city (Q1549591).

queryIsCountry:
A query of type ASK, verifies the subject or URI of a given a literal if it is a type of (P31) sovereign state (Q3624078) or country (Q6256)

queryMonumentCities:
Given the label of an entity of type city, it will retrieve the points of interest (monument) that the city is owner of (P1830) or that is located in the administrative territorial entity of the given city.
It also retrieves the label of type (P131) of the point of interest and its coordinates.
To restrict the search of wanted types of point of interest that the web application wants to display to the user, the label of the point of interest must be at least of one of the following: museum, park, theater, church building, university, fort, sculpture, architectural structure, cathedral, stadium, cultural property, tourist destination. The number of results is limited by default, with the size of 50.



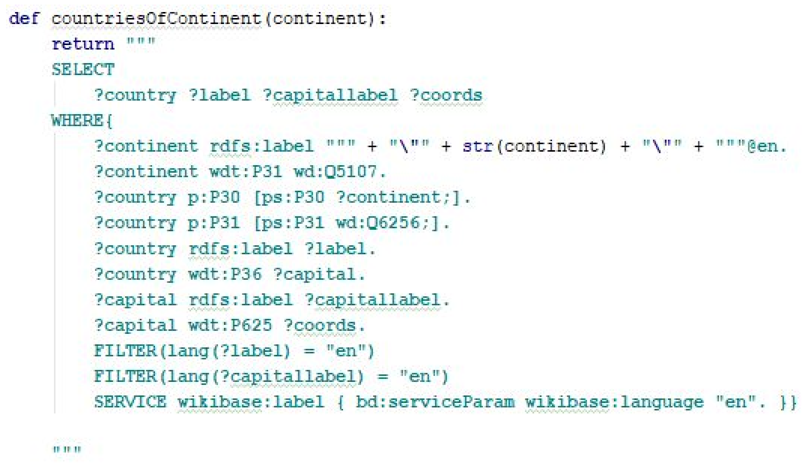
```
def queryMonumentCities(city, limit=50):
    return """
    SELECT DISTINCT
    ?label ?coords ?monuments ?typelabel
    WHERE{
    ?city rdfs:label """ + "\"" + str(city) + "\"" + """@en.
    {
    ?monuments wdt:P131 ?city.
    }
    UNION{
    ?city wdt:P1830 ?monuments.
    }
    ?monuments wdt:P31 ?type.
    ?type rdfs:label ?typelabel.
    ?monuments rdfs:label ?label.
    ?monuments wdt:P625 ?coords

    FILTER(lang(?label) = "en")
    FILTER(lang(?typelabel) = "en")
    FILTER(str(?typelabel)="museum" || str(?typelabel)="park" || str(?typelabel)="theater"
    || str(?typelabel)="church building" || str(?typelabel)="university" || str(?typelabel)="fort"
    || str(?typelabel)="sculpture" || str(?typelabel)="architectural structure" || str(?typelabel)="cathedral"
    || str(?typelabel)="stadium" || str(?typelabel)="cultural property" || str(?typelabel)="tourist destination")
    }limit """ + str(limit) + """
    """
```

queryCoord:
Gives the coordinates of an entity by giving its label.

queryCityCoord:
With the following query it is retrieved the coordinates (P625) of a entity which either has a country (P17,) a capital (P1376), it is located in the administrative territorial entity (P131) or contains administrative territorial entity (P150). The entity is first fetch by the given label. The search is limited by one, so the first resulted item is the only one returned.

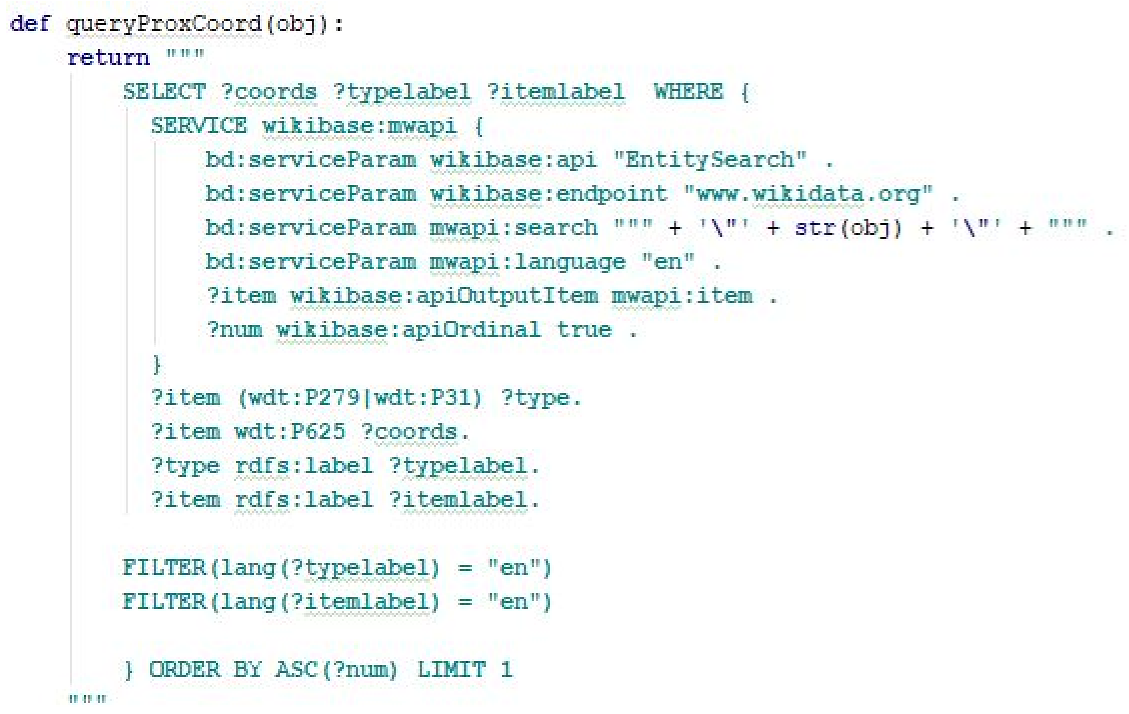
cityWithAirport:
Given the label of a country, it is verified if its entity is an instance of (P31) country (Q6256) (being the ranking of this statement irrelevant on wikidata, this is all the object of such entity and property in matter will be considered). Having an entity of type country, it is gathered the airports of a country whose country (P17) is the entity already mentioned. Having the item of the airport, it’s label, coordinates, city and label of the city are fetched. The results are filtered so that the labels of the airport and city are only written in english.

countriesOfContinent:
Given the label of a continent, the following query will verify if the entity that represents the continent is also a continent on Wikidata (Q5107). The countries item will be gathered by specifying that the correspondent item has as continent (P30) (being the ranking of this statement irrelevant on Wikidata, this is, all the objects of such entity and property in matter will be considered) and also the item must be a type of (P31) country (Q6256).
From the country item, capital through the capital property (P36) is gathered, as well as its label and coordinates. All of the resulting labels are filtered so the application gets the labels in english.


```
def countriesOfContinent(continent):
    return """
    SELECT
        ?country ?label ?capitallabel ?coords
    WHERE{
        ?continent rdfs:label """ + "\"" + str(continent) + "\"" + """@en.
        ?continent wdt:P31 wd:Q5107.
        ?country p:P30 [ps:P30 ?continent;].
        ?country p:P31 [ps:P31 wd:Q6256;].
        ?country rdfs:label ?label.
        ?country wdt:P36 ?capital.
        ?capital rdfs:label ?capitallabel.
        ?capital wdt:P625 ?coords.
        FILTER(lang(?label) = "en")
        FILTER(lang(?capitallabel) = "en")
        SERVICE wikibase:label { bd:serviceParam wikibase:language "en". }}

    """
```

queryCountryOf:
This query finds all entities with a given label and also gets their types. All the resulting entities must have coordinate objects (this is, the entity must have coordinates). The entities are filtered such as the resulting ones has the labels of the type and item in english. The resulting item must have coordinates (P625) and an object as its country (P17). The results are ordered in ascending order and it returns the first result.

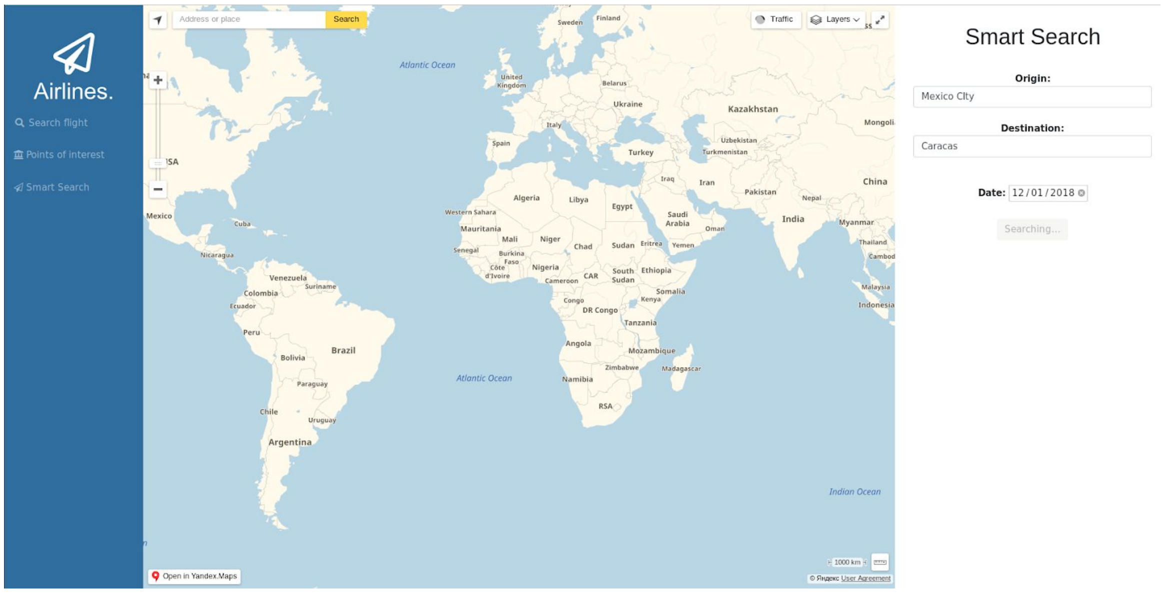
queryProxCoord:
The following query finds all entities with the given label and also gets their types. All the resulting entities must have coordinate objects. It is fetch the label of the type end the item. The entities are filtered such as the resulting ones has the labels of the type and item written in english. The result is limited by one, so the first result is returned to the application.



```
def queryProxCoord(obj):
    return """
        SELECT ?coords ?typelabel ?itemlabel  WHERE {
          SERVICE wikibase:mwapi {
              bd:serviceParam wikibase:api "EntitySearch" .
              bd:serviceParam wikibase:endpoint "www.wikidata.org" .
              bd:serviceParam mwapi:search """ + '\"' + str(obj) + '\"' + """ .
              bd:serviceParam mwapi:language "en" .
              ?item wikibase:apiOutputItem mwapi:item .
              ?num wikibase:apiOrdinal true .
          }
          ?item (wdt:P279|wdt:P31) ?type.
          ?item wdt:P625 ?coords.
          ?type rdfs:label ?typelabel.
          ?item rdfs:label ?itemlabel.

        FILTER(lang(?typelabel) = "en")
        FILTER(lang(?itemlabel) = "en")

        } ORDER BY ASC(?num) LIMIT 1
    """
```

6. Application functionality

6.1 Base interface



Smart Search

Origin: Mexico City

Destination: Caracas

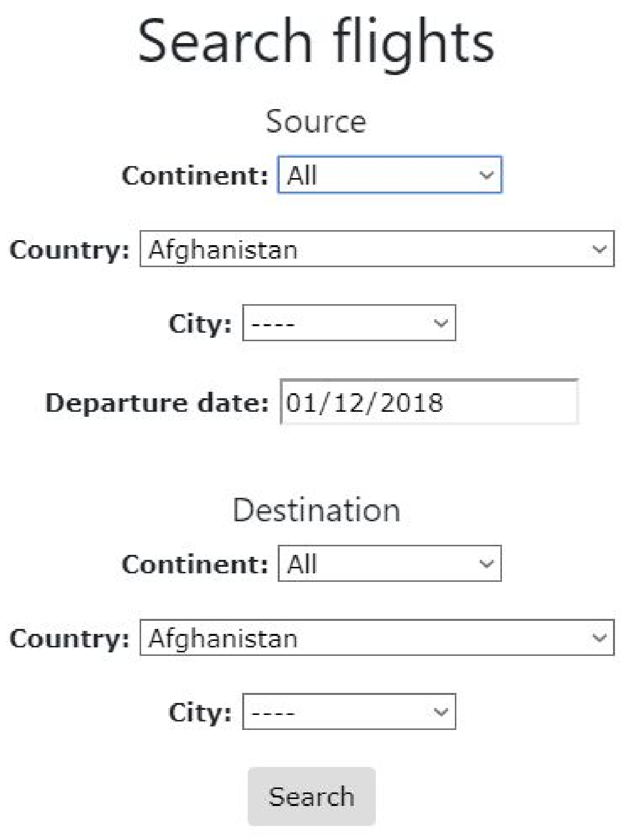
Date: 12/01/2018

Our web application is divided into three main components:

- Left sidebar to navigate to other pages
  - By clicking on another section, the right sidebar will change content and all placemarks on the map will be removed.
- Map where placemarks appear for aiding with visualization
  - Since the Google Maps API is now paid, we chose to use the Yandex Maps API (https://tech.yandex.com/maps/) as it was well documented and also served our needs.
  - The restriction that this choice brought was that we had to write a significant portion of code in JavaScript in order to interact with the map. When there was a need to retrieve data to update the map, we resorted to making a HTTP request to our API to solve that problem.
  - By using the mouse, the view of the map can be changed by dragging it around and zooming in and out with the scroll wheel. Objects placed in the map can be clicked or hovered to show more information.
- Right sidebar with the fields for user interaction with the application
  - Depending on the webpage section, the displayed contents will differ. In general, there will either be selection boxes or text input boxes to choose the desired options and a button for making the request.

6.2 Search flight

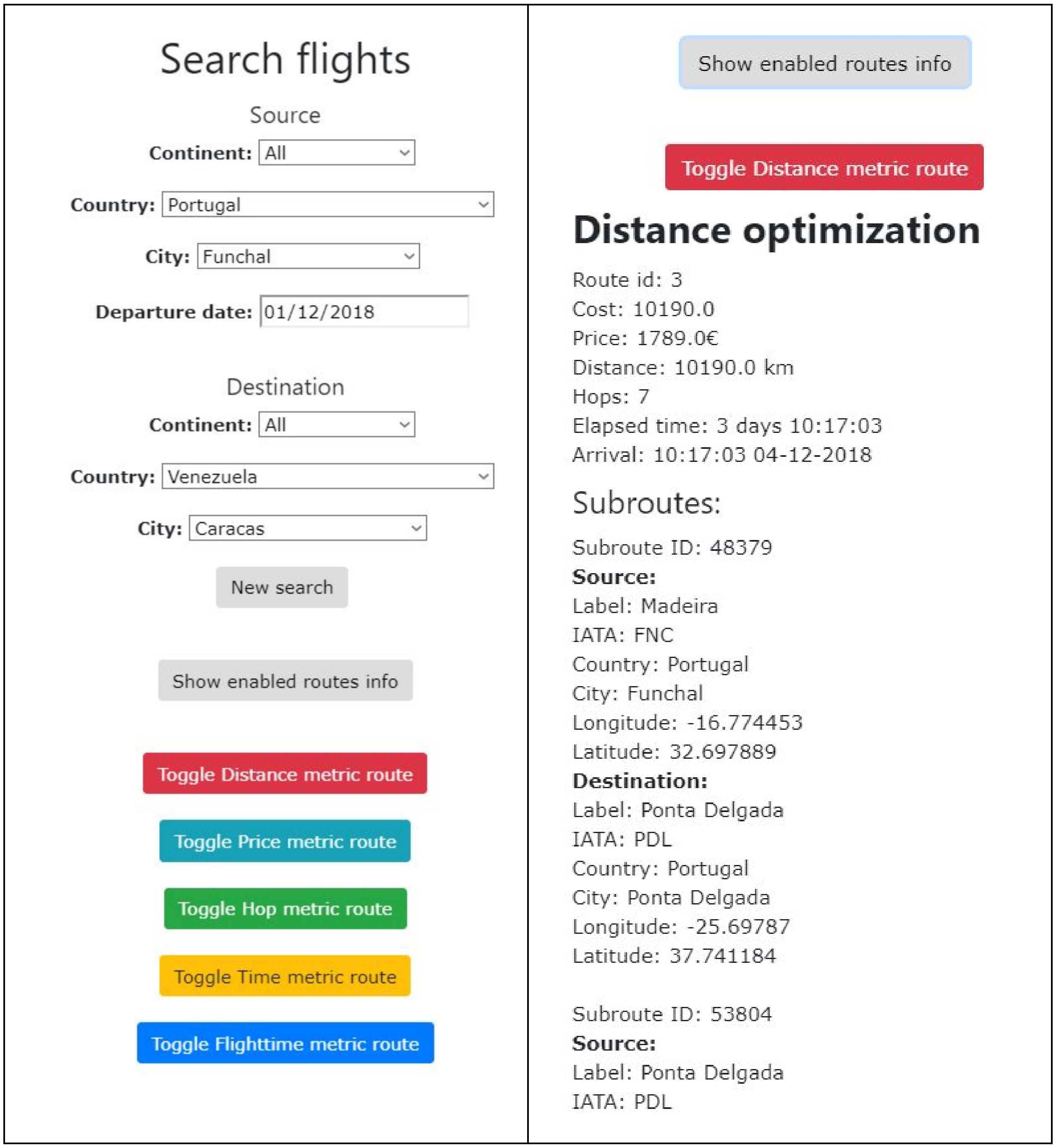
On this section, the user can choose an origin city, a destination city and departure date and see the best routes according to different metrics.



The list of countries can be restricted by continent. The countries are filtered by continent at page load and by selecting a different one, the country listbox will be repopulated with the help of Django templates.
Cities will only show after a country has been selected. Since the number of countries is much larger than the number of continents, the city listbox couldn't be filled like the list of countries as the number of cases are too high to solve it by previously calculating all possible outcomes and using a Django template. Instead, choosing a country triggers a HTTP request to our API which returns the list of cities with airports by country. It is mandatory to select an origin and destination city in order to display the routes.

After clicking on the "Search button", a HTTP request will be made to our API requesting the best routes for the origin/destination cities.
After finishing the query, more buttons will appear on the interface. By default, no route info will be displayed as text or in the map. After clicking in one of the colorful toggle buttons, the

respective route will be shown on the map (clicking it again hides it). The “Show enabled routes info” will show/hide text info about the routes that are visible at the moment (those that had the “toggle” button pressed). The button colors correspond to the placemark colors of the same route. There’s no fixed correspondence between color and type of metric, as it depends in which route was first calculated, so the user needs to pay attention to what metric is written in the button.


Show enabled routes info
Toggle Distance metric route
New search
Show enabled routes info
Toggle Distance metric route
Toggle Price metric route
Toggle Hop metric route
Toggle Time metric route
Toggle Flighttime metric route

Search flights

Source

Continent: All

Country: Portugal

City: Funchal

Departure date: 01/12/2018

Destination

Continent: All

Country: Venezuela

City: Caracas

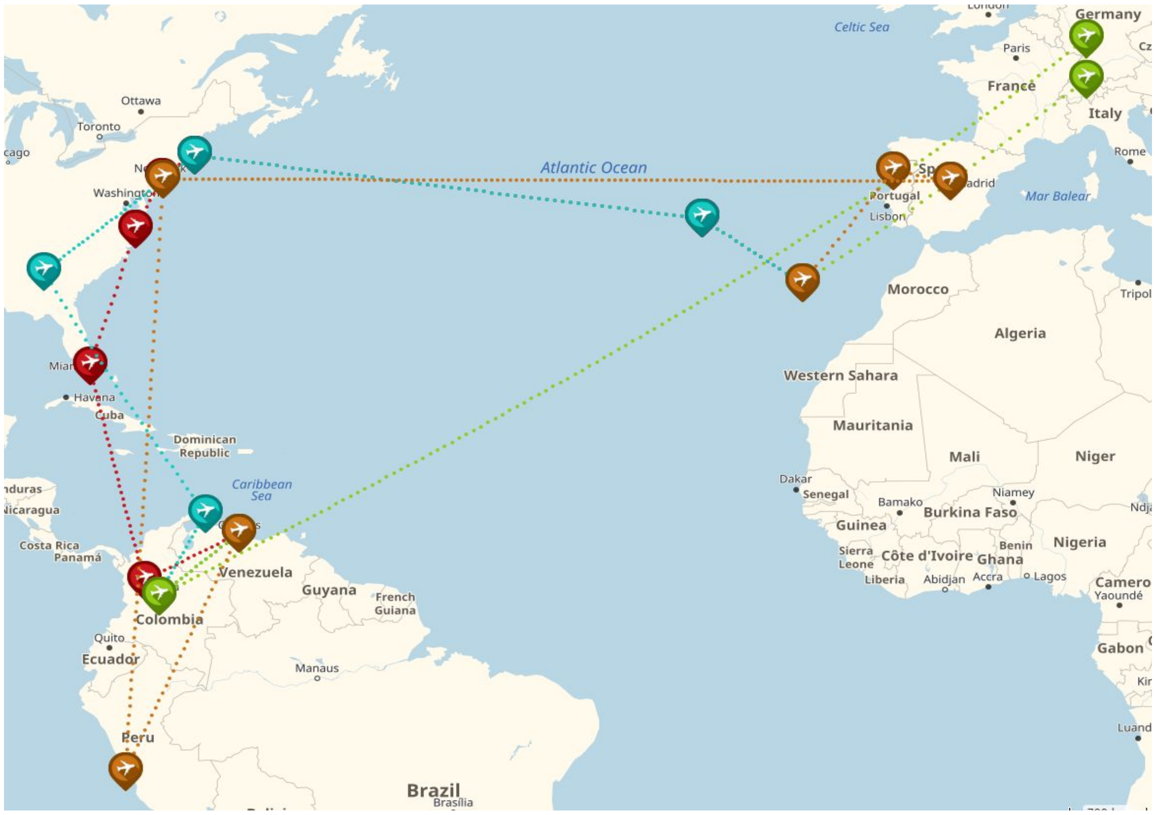
Distance optimization

Route id: 3
Cost: 10190.0
Price: 1789.0€
Distance: 10190.0 km
Hops: 7
Elapsed time: 3 days 10:17:03
Arrival: 10:17:03 04-12-2018

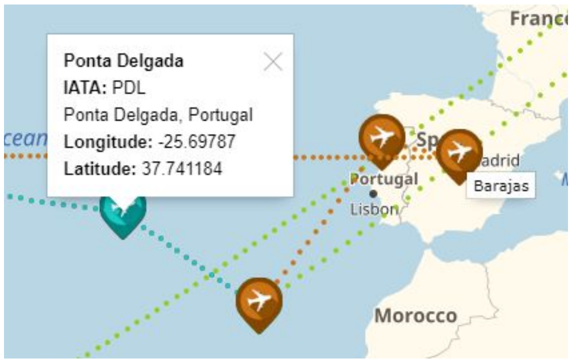
Subroutes:

Subroute ID: 48379
Source:
Label: Madeira
IATA: FNC
Country: Portugal
City: Funchal
Longitude: -16.774453
Latitude: 32.697889
Destination:
Label: Ponta Delgada
IATA: PDL
Country: Portugal
City: Ponta Delgada
Longitude: -25.69787
Latitude: 37.741184

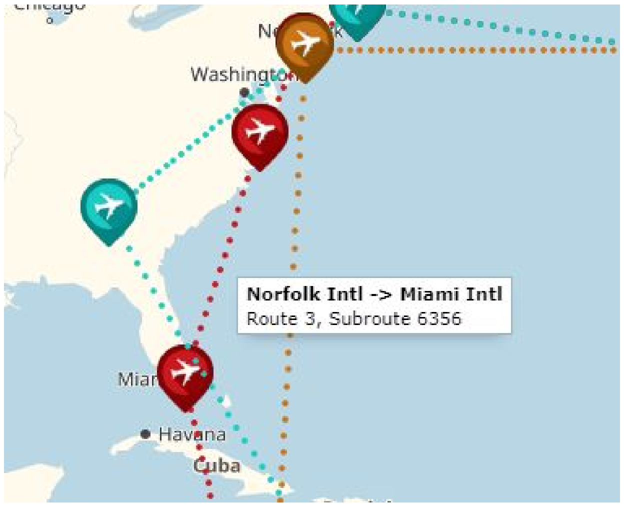
Subroute ID: 53804
Source:
Label: Ponta Delgada
IATA: PDL



In this example, in which a flight was requested between Funchal and Caracas, four routes are toggled to be displayed in the map. In this case, the red icons represent the route with lowest distance value, the green represent the one with least number of hops, the yellow ones are time optimized and blue is for price. When two routes share airports, only one icon will show. Placemarks with a plane symbol are placed on the geographical coordinates of the respective airports. The dotted lines between airports are the subroutes between them.


Ponta Delgada
IATA: PDL
Ponta Delgada, Portugal
Longitude: -25.69787
Latitude: 37.741184

Clicking on an airport placemark will reveal information about it, while hovering over it will only display the airport name. In this example, the airport of Barajas was being hovered.

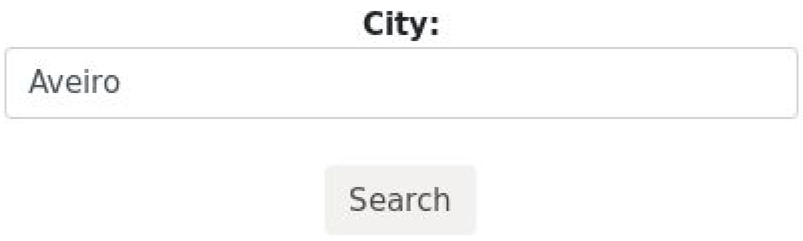


Norfolk Intl -> Miami Intl
Route 3, Subroute 6356

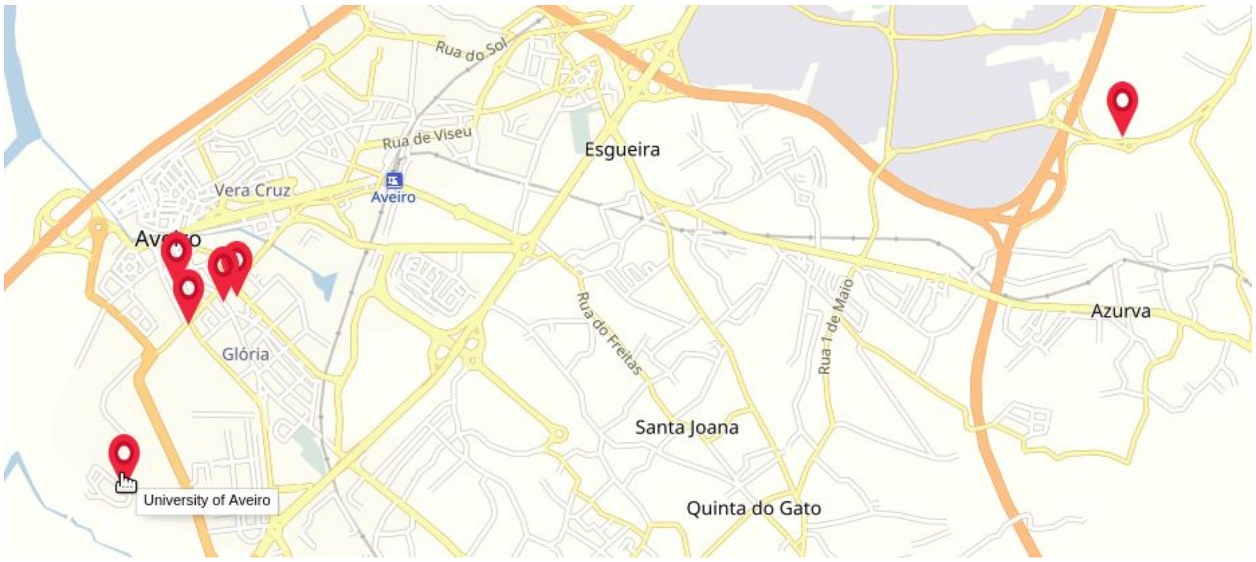
The same way, hovering or clicking on a dotted line will display the route it belongs to and the subroute it represents.

6.3 Points of interest

On the rightmost bar of this section of the web application, the client is able to search a maximum of 50 points of interest of a specific city by writing the name of it on the form pointed by the label 'City'. The client must click on the 'Search' button so the web application can start the search procedure of what it is desired to fetch.

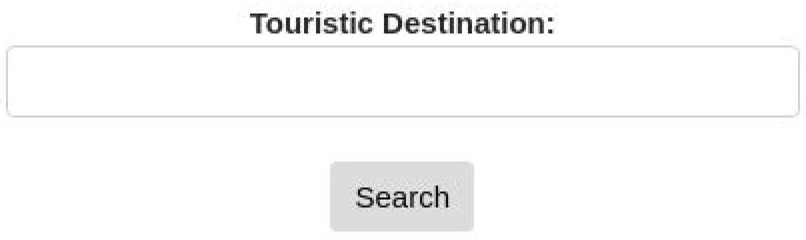


As mentioned earlier on the document, the web application will make of use of the SPARQL query, constructed by the function 'queryMonumentsCities' from the file 'datasets/querywikidata.py', to find the entity by the keywords of the city and later filter all points of interest that matter to display to the user. The data that is gathered by this SELECT query is data stored on the Wikidata servers, so there is a need to use the correct entities and properties to fetch data. The points of interest may be museums, churches, universities, touristic destinations and others. These are pointed on the map by a typical red placemark. By hovering the placemark with the pointer of the mouse, the client is able to see the name attributed to the point, and by clicking on it ,the client will be able to know which type of interest point it is.



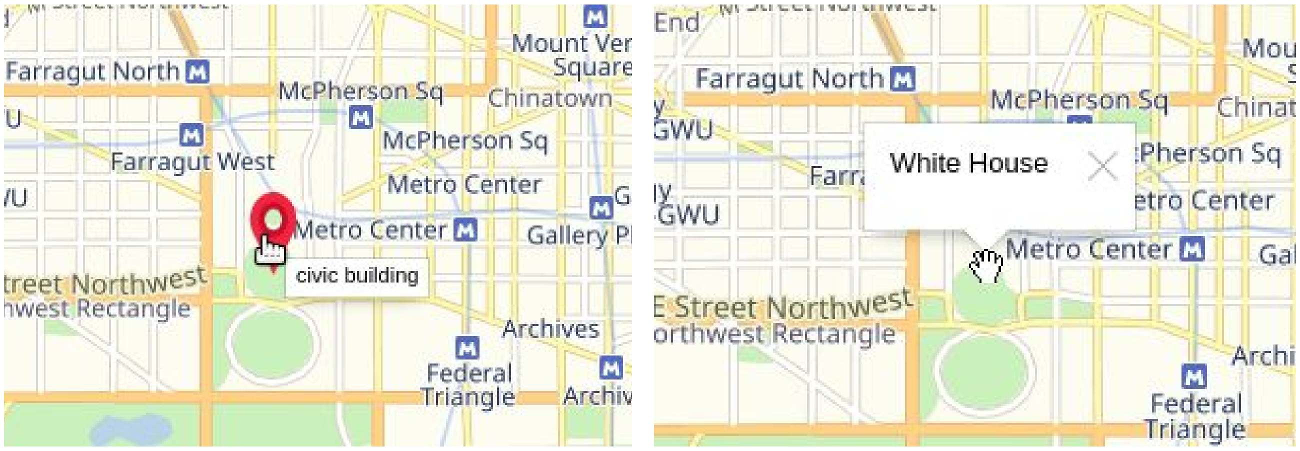
Points of interest of Aveiro

Another functionality is implemented on this section of this website. The client is able to specifically locate a point of interest, or as the web application calls it, a tourist destination. The is able to locate street, parks and all sort of points of interest as long as such entity is registered on Wikidata database. To search the specific point of interest the user must write what it needs to be located on the map in the form pointed by the label "Touristic Destination". To start the data fetching, the client must click on the 'Search' button.



After the submission, the web application will make use of a SPARQL query, constructed on the function 'queryProxCoord' from the file 'querywikidata.py'. Using the input string, the query is executed and the result is an entity which labels matches the inputted keyword. The result will be the complete and correct label of the entity and its coordinates. To notice that if an entity does not have coordinates, the following is not considered, hence not returned.
After getting a result, the web application will point with a typical red placemark the location of the point of interest. By hovering with the pointer of the mouse the placemark, it will display a modal with the type of searched point, and by clicking on it will also show a modal with the name of the point of interest.
On the following example, the user searched 'White House' displaying 'civic building' on a modal as the type of the searched point, and by clicking on it, it will show the name of the

located point, which happens to be “White House”. To notice that it is possible to search points of locations no matter what language that it is written. Because the data is provided by Wikidata, the constructed query must be correctly written with the relevant entities and properties for the search.



Placemark pointing to the White House location

6.4 Smart search

Taking advantage of the functionality of the first two sections of the website, we created the Smart Search section in which with the given name of the source location and destination location, the path between both points is determined, resulting in the best flight routes already mentioned to get to the destination airport but also displaying the path to be taken between the location point and the nearest airport. The flight route is determined between the nearest airport of the source and destination location.

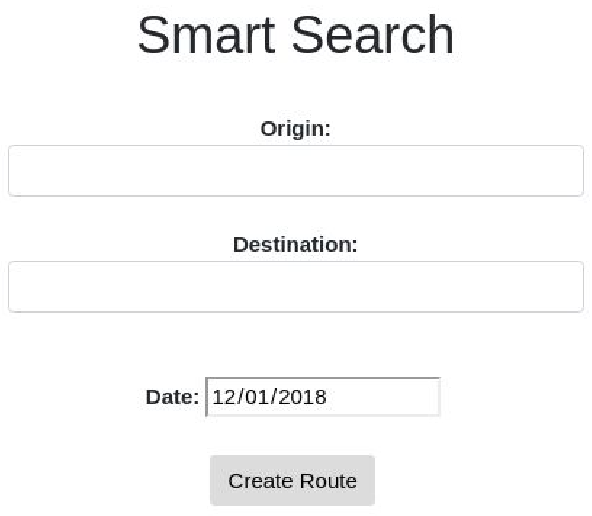
On the rightmost bar, there are three forms pointed to by the correspondent labels:

- Origin: On this form the user defines the departure point of the route. To notice that it could be the name any location, and may not be specifically of an airport.
- Destination: On this form the user writes the arrival point of the route to be established. To notice that the location point may not be the name of an airport.
- Date: On this form the user must specify the date of departure, in order to establish a flight route by considering the date of takeoff of the first plane.

To submit this information to the web application server, the user must click on the ‘Create Route’ button. The information gathered by the server is processed, and the SPARQL queries to search the entity of the locations point by a keyword are constructed. The fetched data comes from Wikidata database, and the result must have a coordinate. This query is constructed in the function ‘queryCountryOf’ that, after the submission, returns the entity of the country and the coordinates of the searched location. Both Origin and Destination location are determined by the mentioned query located on ‘datasets/querywikidata.py’ file.

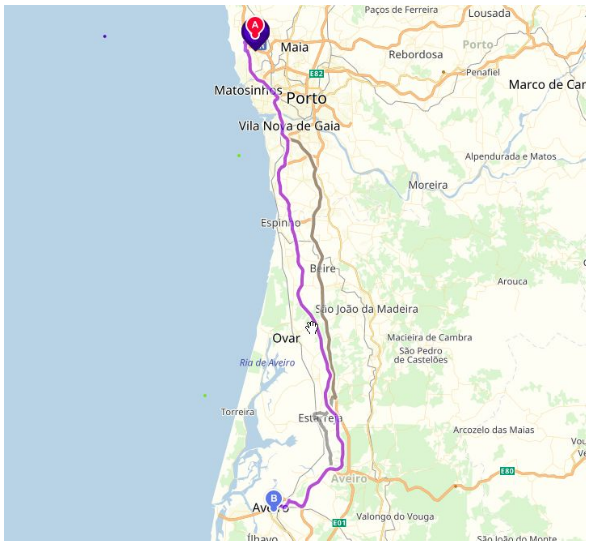
Having the entity of the country, the entities of all airports of the country from the airlinesdot GraphDB database are retrieved. The data is retrieved by using the query that is implemented on the 'getAirportsFromCountry' function. After the query is submitted, the coordinates of all airports from the country of matter are obtained. Having this type of information and the coordinates of the location of the source and destination, the application will determine which of the airports is the nearest from the desired location. This process is also executed for both destination and source location that has been requested by the user.
When the nearest airports from the country of source and destination are determined, the application calculates the best routes from the departure and arrival airports in consideration. This is done by an A* search algorithm already mentioned on this document.
When the information about the locations and best flights routes in terms of different parameters are obtained, the web application through an API gathers the information and process it with the purpose of displaying the placemarks of the destinations points of the pseudo routes of a flight and its territorial routes as well
There are a couple of things to notice:

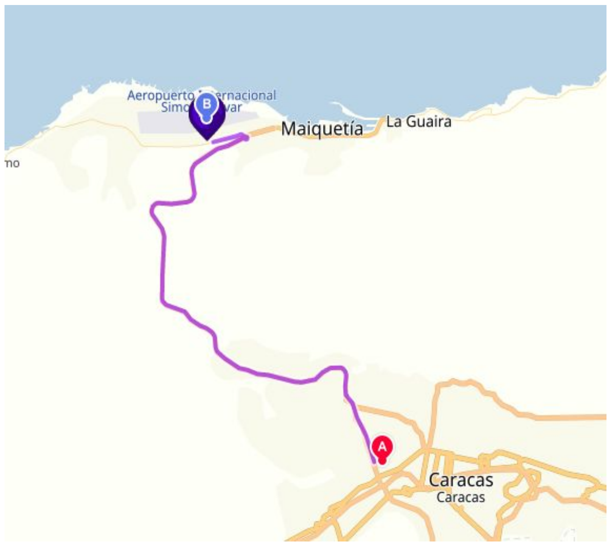
- If either the source and/or destination point are not inserted by the client, the search won't occur, displaying to the user an alert modal specifying the situation.
- If the location of the destiny and/or source can't be found, the user is alerted with a modal about the correspondent situation.
- If one of the location points (the source or destination) is found, then it will only display the territorial routes from the location of interest and the nearest airport. Still, the user is adverted by a modal notifying it about the situation in matter.



On the following example, the user searched flights to travel from "se de aveiro" to "Caracas" on 12/01/2018.

Airlines chooses “*Aeroporto Francisco Sá Carneiro*” as the nearest airport of “*se de Aveiro*”. It also displays the route that needs to be taken by car to get to the mentioned airport.
Moreover, the web application chooses “*Aeroporto Internacional Simón Bolívar*” as the nearest airport of Caracas. Similarly, the path by car to the correspondent airport is displayed.





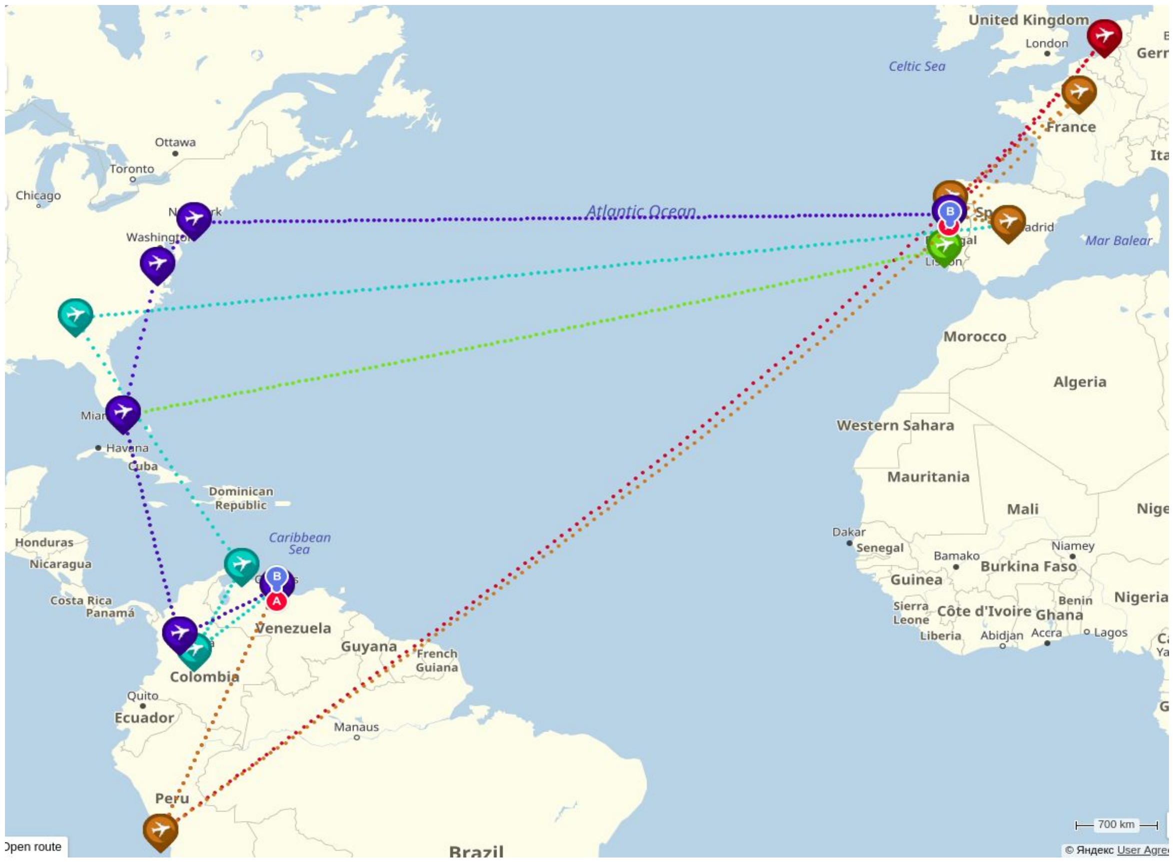
On the left: Territorial routes from “Se de Aveiro” to “*Aeroporto Francisco Sá Carneiro*”
On the right: Territorial routes from “*Aeroporto Internacional Simón Bolívar*” to “*Caracas*”

The routes of the flight from the source airport to the destination airport are displayed as well. To notice that the colors of the routes and placemarks represent a type of best flight. The meaning of the colors in this context are:

- Red - Best flight in terms of number of routes.
- Light Blue - Best flight in terms of price.
- Green - Best flight in terms of distance to travel.
- Orange - Best flight in terms of time of travel from the source to the destination.
- Blue - Best flight in terms of the somatory of the time spent on the taken flights.

By hovering with the pointer of the mouse on the path of a flight, the user can notice the total price, distance and duration of the flight. By clicking on it, the user is able to see the source and destination airports of the pseudo-flight.

By hovering with the pointer of the mouse on a certain placemark of a flight route, the user is able to see the name of the airport. By clicking on it, information about the IATA, name of the location, longitude and latitude are displayed by a modal.



Flight Routes from “*Aeroporto Francisco Sá Carneiro*” to “*Aeroporto Internacional Simón Bolívar*”

7. Conclusions

We ended up taking full advantage of all the technology that needed to be used to create a Web Application that is capable to help the user find out the best flight between two points. For this project, we learnt how to fully work with RDF, more specifically with N-Triples and Notation 3. This project was also a chance for us to reuse knowledge from other disciplines such as IIA (Introdução à Inteligência Artificial). Furthermore, we worked with GraphDB Triplestore as a repository to the required data. To select and update the triplestore data of the repositories, we used SPARQL and to publish semantic data about the flights and its routes we used RDFa. The technology stack was relevant to process our dataset.
We used this technology to efficiently gather relevant information and making use of it, approaching other areas of Informatics in a way to determine the best flight routes efficiently.
Django simplified many aspects of the development that would have otherwise made it very difficult to focus on the essential part of this project.
Websites like Wikidata provide a good source of normalized data that can be easily used to a numerous sort of projects, since the amount of entities and relations it contains is so vast, although sometimes it lacked coherence between similar items.
Implementing this project helped us better understand in which situations storing data as graphs can be useful, as well as the limitations it brings. It's a fact that the web would benefit significantly from a more standardized semantic approach to data, making it possible for common computer applications to process data in a pragmatic way without requiring human intervention or validation.

8. Instructions to run the application

8.1 Preparing the environment

Our development environment was based on Linux (Arch Linux and Ubuntu). While trying our application on Windows, we didn't run into any problems, but since we didn't test it as thoroughly, we cannot guarantee it fully works.
To be able to run our application, one must have python3 and GraphDB installed.
The python packages needed to run our project are rdflib, sparqlwrapper, s4api and django. If those requirements aren't installed, the following command can be run to install them:

```
$ pip install rdflib sparqlwrapper s4api django
```

8.2 Preparing the data

After installing everything, the next step is to create the databases on GraphDB and load the data.
Optionally, it's possible to regenerate data by running the converter script explained at the end of the section 2.1.
In GraphDB, create a new database called 'airlinesdot' and another called 'airlinesdotCache'.
Import the N3 file 'datasets/airlinesdot.n3' to the 'airlinesdot' database.
The project should now be ready to run.

8.3 Launching the project

Firstly, the GraphDB server needs to be launched.
After that, the django project can be launched:

```
$ python manage.py runserver
```

8.4 Admissible data

As mentioned before, due to some limitations in our dataset, not all airports are possible to reach. With some knowledge on which airports admit commercial flights a compatible city is easy to choose, but to simplify the testing and validation of our application we will list some cities that are guaranteed to work when searching for flights. To keep in mind this isn't an exhaustive list, but a set of values that's almost guaranteed to function when combined.

Angola ● Luanda Brazil ● Belo Horizonte ● Curitiba ● Rio de Janeiro ● Sao Paulo Bulgaria ● Sofia ● Canada ● Toronto ● Vancouver Cuba ● Havana Costa Rica ● Nicoya Denmark ● Billund ● Copenhagen Ecuador ● Quito Egypt ● Alexandria	El Salvador ● San Salvador Finland ● Ivalo France ● Lyon ● Marseille ● Nice ● Paris Germany ● Berlin ● Frankfurt ● Munich Ghana ● Accra Greece ● Thessaloniki India ● Ahmedabad ● Agartala ● Mumbai ● Port Blair Iraq ● Erbil	Italy ● Genoa ● Milano ● Venice Jamaica ● Kingston Japan ● Akita ● Fukushima Latvia ● Riga ● Ventspils Madagascar ● Antananarivo Mexico ● Chihuahua ● Mexico City ● Cancun Namibia ● Walvis Bay Netherlands ● Amsterdam ● Rotterdam

New Zealand • New Plymouth Mozambique • Maputo Poland • Warsaw • Wroclaw • Katowice Portugal • Porto • Lisbon • Faro • Funchal • Ponta Delgada	Russia • Moscow • Vladivostok Slovakia • Bratislava South Africa • Cape Town • Port Elizabeth Spain • Barcelona • Bilbao • Cordoba • Granada • Ibiza • Leon • Madrid • Canarias	Switzerland • Zurich United Arab Emirates • Abu Dhabi United Kingdom • Birmingham • Leeds • London • Southampton Venezuela • Caracas • Merida Zimbabwe • Bulawayo

This issue also affects smart search, since it might find that the nearest airport is one of those that cannot be reached. The following table contains some tested examples of points of interest:

Australia: • Ayers Rock Egypt: • Grande Esfinge France: • Lascaux Italy: • Torre de Pisa Japan: • Suzuka circuit	New Zealand: • Mount Eden Portugal: • Autódromo Internacional do Algarve • Autódromo do Estoril • Costa nova • Sé de aveiro • Palácio da Pena • Praia da rocha • Universidade de Aveiro	Scotland: • Lago Ness Spain: • Guggenheim Bilbao