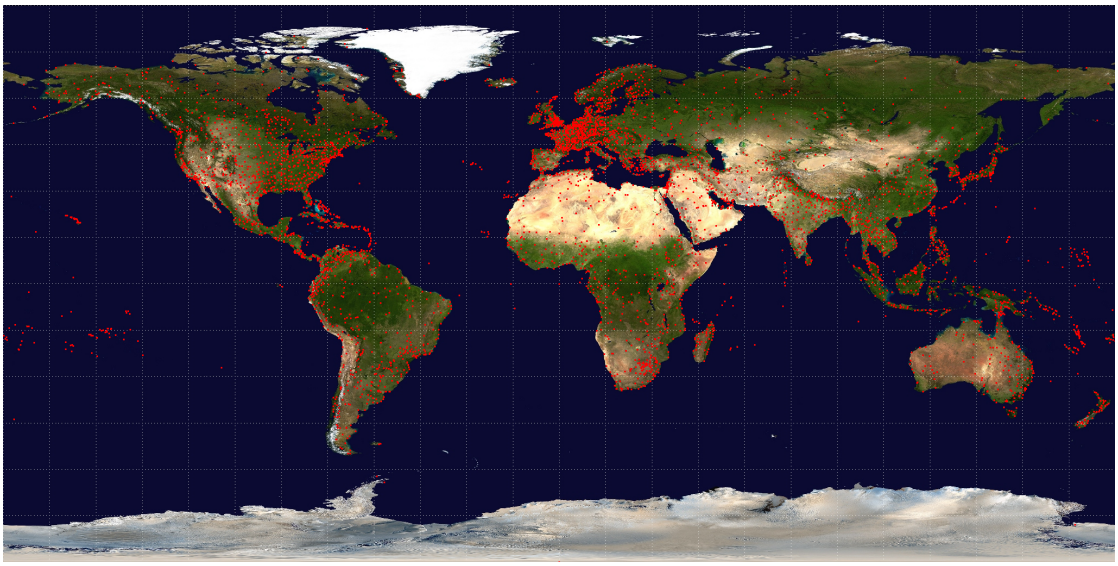
Information Retrieval
Lab 03

Page-Rank implementation

Badouh, Asaf
Cebollero, Laura

6th of November, 2018



1 Introduction

In this lab we want to compute the page rank of airports using the network defined by routes from and to airports. For that, we used the supplied data downloaded from *Open flights*[1], which has been provided to us:

- `airport.txt` - contains a list of airports (**nodes**) from the world.
- `routes.txt` - contains a list of routes (**edges**) from the world.

As explained in the lab statement, we can represent this information as a graph, where airports are nodes and routes are edges. The **weights of the edges** will be the **number of flights between two airports**. Thus, **airports** with a **high PageRank** value are ***important* airports**.

This is crucial information, for example, when designing and reforming airports, as it means that many flights (either with passengers or goods) visit it when landing or taking off. In other words, they may be affected by such remodellings.

Optimizing the operations of such important airports will lead to improvement of many supply chains and other visitors.

2 Implementation Scheme

Having described and seen the problem at hand, we will focus now on how the implementation of the airport has been done. We have chosen to stick to Python for this project.

Since some base code has been provided to us, we will focus mainly on two aspects:

- The **data structures** we have settled on after implementation for both the nodes and edges.
- The **difficulties** we have faced, as well as the **decisions** taken to face them.

2.1 Data Structures

As described on the introduction, there are only two main data structures:

- The nodes, which act as Airports.
- The edges, which represents the *directed* routes between two airports.

To create the nodes we have read them from the file `airport.txt`.

While analyzing it and creating our data structure, we have found a total of $7,663$ airport records. However, we have found some special cases:

1. From the total, $1,923$ of them don't have IATA code, which is what we are using as an identifier for the airport.
2. There are two airports that appears twice, so they are listed twice on the file.

To avoid having duplicates and working with airports with a missing identifier, we have discarded them. **This has left us with a total** $5,738$ ***valid* airports**.

[1] More details about the data structure can be found in the lab: Session03

If we were to represent the problem in a two-dimensional array, that is, a matrix, it would consume a large size of bytes. Namely:

$$size =\sim 2^{25} \times sizeof(datatype)$$

Since the maximum weight found is 534 (ORD - Chicago Ohare Intl, United States) our datatype must be at least 2-bytes, *i.e. total of 64 Mega-Bytes.*

This hypothetic **matrix would be very sparse**, so in order to save data space [2], we have decided to save the information in two dictionaries, one for each structure: airports and routes.

Below, in figure[1], you can see how the structures have ended up being defined.

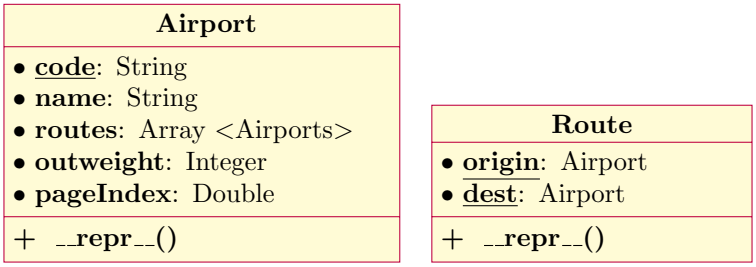


Figure 1: Class diagrams

Each attribute of the **airport** consists of:

- **code**: IATA airport code. Identifier of the airport.
- **name**: Airport name.
- **routes**: List of all airports that have a route that points to the current airport. In other words, list of airports that have as a destination this airport.
- **outweight**: Number of departure flights from this airport.
- **pageIndex**: Page-rank index.

As for the Route structure, its structure is way simpler:

- **Origin**: The airport from where the flight takes off.
- **Dest**: The destination airport, where the flight will land on.

The `repr` function in both structures just defines how the airport and route should be defined.

3 Difficulties and decisions taken

Implementing the code and understanding the algorithm was pretty straightforward. However, we have faced several problems:

[2] In fact, 64MB is not that big nowadays, in efficiency terms it might even be better to save the data as matrix in order to save data manipulation and random accesses. Nevertheless, we have chosen to stick to the skeleton structures provided.

- **Duplication of airports as well as airports without IATA code** - This has been thoroughly explained in the Implementation section.
- **Treatment of dangling nodes** - That is, treatment of airports that are acting as destinations, but not as origin of routes, resulting in $outweight = 0$.
- **Pagerank for dangling nodes** - Somewhat affected by the second point.
- **Proper number of iterations** - Constant number vs. Convergence.

3.1 Dangling nodes

Our main difficulty was dealing with the *airports* without departures, that is, the dangling nodes. There are three accepted approaches for treating nodes with no outgoing edges[1]:

1. **Eliminate** such nodes from the graph (iteratively prune the graph until reaching a steady state).
2. Consider such nodes to **link back** to the nodes that link to them.
3. Consider such nodes to **link to all the other nodes** (effectively making an exit out of them equivalent to a random jump).

The first approach can lead to loss of information. Consider the graph in figure [2] that represents our network. If we prune the graph after 2 iterations we can see that we have lost a lot of information, namely in this case a **total of 3 airports**. So we have discarded this option.

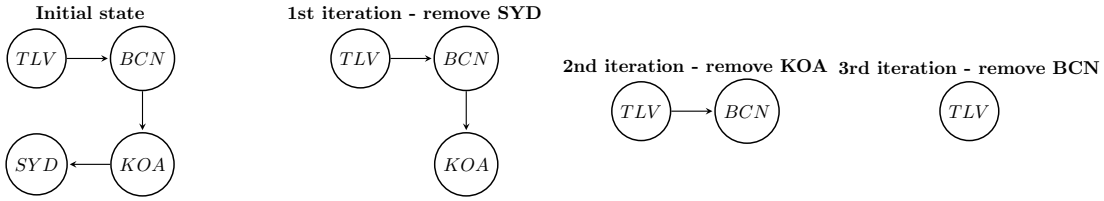


Figure 2: 1st approach

The second and third approaches are more robust and ensure not losing information. However, applying the second option would be creating false routes that are, in fact, not real, since the graph is directed for a reason: taking into account the direction of routes. In other words, by applying it we would be biasing the data too much by creating new routes that do not exist, thus the resulting pagerank would be incorrect.

This leaves us with the third option, which is the more reasonable one with the problem at hand. W**e have decided to add edges to all the disconnected nodes to take into account random jumps** (for example, changing the route when doing an scale in an airport).

As implementation detail it worth mention that we don't keep this new edges on the data structure itself, but we have instead added it to the iterative computation of the pagerank algorithm. Therefore, effectively not having to store n^2 new edges, where n is the number of dangling nodes.

3.2 Pagerank for dangling nodes

Although normalizing weights by itself was not difficult for the *normal* nodes (i.e. nodes that were not dangling), we had to consider how to compute the pagerank for the dangling nodes taking them as a special case on each iteration.

This ended up by treating their pagerank separately with the well-connected ones:

1. Let d be the number of dangling nodes. That is, the nodes with outweight 0.
2. Let n be the total number of nodes.
3. $prDangNodes = L * \frac{d}{n}$
4. $evDangNodes = \frac{1}{n}$
5. for each iteration
   - $evDangNodes = prDangNodes * evDangNodes + \frac{1-L}{n}$

4 Experiments

The last step of this project has been performing some experiments.

4.1 Convergence

During the implementation of this project, we started with a small number of iterations: only 10.

After ensuring that the pagerank algorithm was working correctly, we then addressed the suggestion of using a convergence parameter to act as a flag to stop computing the pagerank after reaching a threshold in terms of difference between one pagerank vector P and the new computed one.

The threshold stablished is of value $1e^{-8}$. This convergence has led us to a total of 14 iterations. If we check the convergence factor for each iteration, tho, we can see that we are working with very small units:

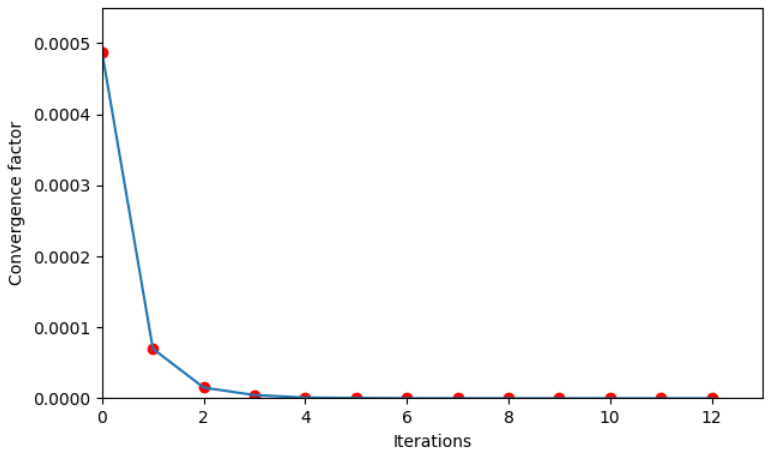


Figure 3: Convergence factor diff. for each iteration

4.2 Different dumping factors

TODO

4.3 Efficiency

As a side note, we have noticed that our algorithm is not as fast as we would like it to. With the convergence flag (thus, 14 iterations) it takes a total (on average) of 13 seconds. Almost a second per iteration!

Using the library `cProfile` we have pinpointed our problem: using the structure provided by the problem, we have a bottleneck in times of execution when computing the overall sum of the pagerank for a given airport.

For context:

```
1  def computeSumDestVert(P, n, i):
2      overallSum = 0
3
4      # Iterate through routes that have i as destination
5      for e in airportList[i].routes:
6          j_code = e.origin
7          w_j_i  = e.weight
8          airport_j = airportHash.get(j_code)
9          j = airportList.index(airport_j)
10         overallSum += P[j] * w_j_i / airport_j.outweight
```

In line 9 we have the bottleneck. We are getting the index of an airport to specify its position on the vector of pageranks P. However, to get this index we need the whole object, which is what we are retrieving on line 8.

This is caused by the structure used. If we were using an adjacency matrix, this problem could be avoided, since we would be working with indexes directly.

Although the bottleneck is minimal with the current data, for future work it would be useful to reimplement this part using a more efficient structure or strategy.

5 Conclusions

The conclusions we have arrived to after doing this project are manifold:

- The iterative algorithm for pagerank is effective and somewhat efficient for the data at hand. A future work could check the scalability: how efficient would it be with a billion nodes and million nodes? For example.
- **Dangling nodes have to be taken into account** and affect a lot pagerank. We didn't take them into account on our first pagerank computations and thus, the sum of all pageranks did not amount to 1.

- Choosing an right solution for the dangling nodes should be top priority. In our case since the number of nodes was not that large we could opt for the solution of linking all nodes. However, there may be easier cases (computationally speaking) where linking back the nodes to the ones they are coming makes sense.
- Using a convergence of $1e^{-8}$ leads in our case to a total number of 14 iterations, which is not that far from what we started on: 10 iterations.

References

[1] Yahoo Labs Ronny Lempel. Introduction to search engine technology. https://webcourse.cs.technion.ac.il/236375/Winter2013-2014/ho/WCFiles/lec2-linkAnalysisIntro.pdf, Winter 2013-2014.