PageRank

Josep de Cid Rodríguez, Adrián Sánchez Albanell

November 2018

Abstract

In this practice, we are going to implement a PageRank system similar to how web searches work, but using airports and routes as a simile to the webs and links among these.

1 Plan of attack

Inspecting the contents of the files `airports.txt` and `routes.txt`, we observed that most of the data columns are useless for our needs, so we decided to remove them in a previous process to speed-up the real page-rank calculus.

From the airports file we will only keep the IATA code, name and country. Also, as the session statement says, we removed all airports with invalid IATA codes or without one, which left us 5629 valid airports of 7663.

From the routes file, we only need to keep the origin and destination for each route

Here we have the pseudo-code algorithm provided in the statement:

1: n = number of vertices in G;
2: P = any vector of length n and $sum1$ (for example, the all $\frac{1}{n}$ vector);
3: λ = the chosen damping factor, between 0 and 1;
4: **while** not stopping condition **do**
5: Q = the all-0 n-vector;
6: **for** i in 0..n-1 **do**
7: $Q[i] = \lambda \cdot sum \left\{ \frac{P[j] \cdot w(j,i)}{out(j)} : \text{there is an edge } (j,i) \text{ in } G \right\} + \frac{1-\lambda}{n}$;
8: **end for**
9: $P = Q$;
10: **end while**

Before starting the implementation we are going to define each part explicitly, mostly constant or user-defined behaviours, like the damping factor λ, defining the stopping condition, the initial PageRank vector...

- $\boldsymbol{P_0}$: As an iterative algorithm, we need to provide a initial vector for iteration 0, defined at line 2. This initial vector will be the all $\frac{1}{n}$ vector. After that we will try different initial PageRank vectors (Section 3), concluding that all convergence to the same values with more or less iterations.

- $\boldsymbol{P_i}$: In the iteration i we will reassign the new calculated rank to the current airport rank (line 9). We will repeat this operation until the stopping condition isn't satisfied.
- **Damping factor $\boldsymbol{\lambda}$**: In PageRank, it represents the probability that the user keeps navigating via links or jumps to any node equiprobability. For this initial implementation we will set $\lambda = 0.8$, as the statement suggest. Later on we will try different values of λ and see how it affects when the value is not close to 1.
- $\boldsymbol{P_i[k]}$: The new rank calculus for each airport is provided by the following formula $\lambda \cdot \sum_{j \to k} \frac{P_{i-1}[j] \cdot w(j,k)}{out(j)}$ (calculating $out(j)$ as the sum of all outgoing edge weights of j) plus the equiprobability $\frac{1-\lambda}{n}$.
- **Stopping condition**: We will define two conditions to stop the iterative method, by setting a max number of iterations (`MAX_ITERATIONS=1000`) and the convergence criteria, achieved when for each airport, score between the current iteration and the last one, if this error falls below a certain threshold. The error rate of a node i is defined as the difference between the score of the vertex $P(V_i)$ and the score computed at a certain iteration k, $P_k(V_i)$. Since the real score is not known *a priori*, this error rate is approximated with the difference between the scores computed at two successive iterations and checking that is lower than an epsilon threshold, that we will set at $0.5 \cdot 10^{-2}$:

$$abs(P_{k+1}(V_i) - P_k(V_i)) < \epsilon$$

2 Implementation

To start we are going to modify the given class Airport to simplify and do the process more efficiently, and also remove the Edge class, which even being a class representable concept, we consider it useless because we can do the same job by simply using maps more efficiently.

In the class Airport we are going to define the following new instance attributes, `rank` and `new_rank`, to store directly the ranks for each iteration in the proper instances and avoid to have a confuse and overloaded indexing system which is also not efficient to be used alongside with hash dictionaries.

We will also set only one routes related attribute, a dictionary used as a hash being the key the origin airports i for all routes from the airport i to the given one and the value, the weight of that route (number of routes from j to our airport).

```
class Airport:
    def __init__(self, iata=None, name=None):
        self.code: str = iata                # Airport IATA code
        self.name: str = name                # Airport name + country
        self.routes: Dict[str, float] = {}   # Routes: {origin_airport: weight}
        self.out_weight: int = 0             # weight of outgoing edges
        self.rank: float = 0.0               # actual rank
        self.new_rank: float = -1.0          # new rank
```

First, let's read the airports data into the proper data structures. For each line in the file, we create an instance of Airport with the proper values at `code`, `name` and index it into a

`dict` structure being the IATA code the key. The time cost of this process is $\Theta(n)$ considering constant the time to create an Airport instance.

Let's now read the routes having the previous airports hash structure already filled. For each route, lets filter routes with invalid IATA codes, and the ones which its origin or destination is not in the airport's list, as the deliverable says. The remaining routes can be introduced in the destination airport instance into the attribute routes, adding 1 to the weights to the corresponding origin airport key. While iterating we will also increase the value of the out weights for each origin treated. The time cost of this process is $\mathcal{O}(m)$ being $m = |Routes|$ and being able to access airport hashes in constant time.

There's one problem that we will face while computing the page rank, the sink nodes, dead ends with no outgoing links such that the probability that multiplies λ is lost. The solution is to add a factor S such that we change the second term into

$$\frac{1 - \lambda + \lambda S}{n}$$

such that S = sum over PR of sinks [1]. By creating the list of sink nodes before stating in $\Theta(n)$ we can access to its ranks later very efficiently.

The total cost of preprocessing the files and filling data structures is $\mathcal{O}(n + m)$.

To compute *PageRanks* let's first initialize all airport's initial values at $1/n$ in time $\Theta(n)$. Now while the bool condition is not satisfied, let's compute the sum of *PageRanks* for the sink nodes in $\mathcal{O}(n)$. For every airport we calculate its new rank by accessing its incoming neighbours ($\mathcal{O}(m)$) using already calculated values in constant time. We know the number of edges m is $\mathcal{O}(n)$, not $\mathcal{O}(n^2)$, so we are going to go through any edge just once, given a total cost of $\mathcal{O}(n + m)$.

The iterative process is going to be repeated a small constant number of times `MAX_IT` so the total cost of the computation will be lineal in $\mathcal{O}(n + m)$ parametrized by `MAX_IT`, that is also the final cost of the whole algorithm.

3 Experimenting and results

The two parameters we can experiment with to check how our results change are the chosen damping factor (λ) and how we initialize our rank vector. To initialize the vector we have chosen three different methods:

- **All $\frac{1}{\mathbf{n}}$:** all the ranks are initialized at $\frac{1}{n}$, where **n** is the number of airports.
- **Just one one:** all the ranks initialized at 0 but one that is initialized with value 1.
- **$\sqrt{\mathbf{n}}$ $\frac{\mathbf{1}}{\sqrt{\mathbf{n}}}$'s:** $\sqrt{n}$ values initialized at $\frac{1}{\sqrt{n}}$ the remaining values at 0.
- **Random normalized:** all values initialized at random and then normalized.

It is trivial to see that the sum of all the ranks is 1 in each initialization. For the damping factor we will just try different values from 0.9 to 0.5 with steps of 0.1.

From the table 1 we can observe that by changing the value of λ the execution time and number of iterations are reduced considerably, given that the algorithm converges faster. But

this implies that we get worst results as seen in table 2. MAE (Mean absolute error) may seem negligible because PageRank values are in $[0, 1]$ and their sum must be 1, so each value is very low, and because of that MAE is also.

In table 3, where we also use MAE to compare the different initialization vectors with the same λ, we can see that it doesn't affect the results.

Different vector initializations give no difference at all, so it's an empirical prove that all vectors converge to the same vector. Different costs and iterations are maybe produced by not controlled factors as CPU, Ram... (table 1). Given that the random initialization took much more time due to random process overheat, and all vectors lead to the same result is not worth to use this one as the initial choice.

To conclude with, we can extract two main points for our experiments:

- **Initial vector** for PageRank isn't important due to all vectors converge to same value with more or less iterations.
- As we decrease $\boldsymbol{\lambda}$ we get the results quicker, but on the other way, we lose precision and get more inaccurate results.

Here is a sample for our Top 5 ranked airports using the parameters described in section 2:

```
Denver Intl, United States (DEN): 0.005175759980813796
Chicago Ohare Intl, United States (ORD): 0.005063828355606924
Los Angeles Intl, United States (LAX): 0.005009777429274034
Hartsfield Jackson Atlanta Intl, United States (ATL): 0.0039444420275351055
Sydney Intl, Australia (SYD): 0.0037895147318080633
```

	$\lambda = 0.9$		$\lambda = 0.8$		$\lambda = 0.7$		$\lambda = 0.6$		$\lambda = 0.5$	
	t(s)	#it	t(s)	#it	t(s)	#it	t(s)	#it	t(s)	#it
All $\frac{1}{n}$	1.30	97	0.61	46	0.39	29	0.28	20	0.20	15
Just one one	1.19	90	0.69	52	0.48	37	0.36	28	0.30	22
$\sqrt{n}$ $\frac{1}{\sqrt{n}}$'s	1.14	87	0.64	48	0.45	34	0.33	25	0.27	20
Random normalized	3.30	96	1.61	47	1.06	31	0.77	22	0.52	15

Table 1: Execution time and iterations done for different initialization and λ values.

	$\lambda = \mathbf{0.9}$	$\lambda = 0.8$	$\lambda = 0.7$	$\lambda = 0.6$	$\lambda = 0.5$
All $\frac{1}{n}$	0	$3.87187 \cdot 10^{-5}$	$6.79333 \cdot 10^{-5}$	$9.25402 \cdot 10^{-5}$	$1.14050 \cdot 10^{-4}$
Just one one	0	$3.87189 \cdot 10^{-5}$	$6.79335 \cdot 10^{-5}$	$9.25404 \cdot 10^{-5}$	$1.14051 \cdot 10^{-4}$
$\sqrt{n}$ $\frac{1}{\sqrt{n}}$'s	0	$3.87189 \cdot 10^{-5}$	$6.79334 \cdot 10^{-5}$	$9.25403 \cdot 10^{-5}$	$1.14051 \cdot 10^{-4}$
Random normalized	0	$3.87187 \cdot 10^{-5}$	$6.79333 \cdot 10^{-5}$	$9.25402 \cdot 10^{-5}$	$1.14051 \cdot 10^{-4}$

Table 2: MAE between different λ values, comparing each one with $\lambda = \mathbf{0.9}$.

	All $\frac{1}{n}$	Just one one	$\sqrt{n}$ $\frac{1}{\sqrt{n}}$'s	Random normalized
$\lambda = 0.9$	0	$5.83 \cdot 10^{-10}$	$4.63 \cdot 10^{-10}$	$1.23 \cdot 10^{-11}$
$\lambda = 0.8$	0	$3.83 \cdot 10^{-10}$	$3.33 \cdot 10^{-10}$	$2.56 \cdot 10^{-11}$
$\lambda = 0.7$	0	$1.89 \cdot 10^{-10}$	$1.84 \cdot 10^{-10}$	$3.16 \cdot 10^{-11}$
$\lambda = 0.6$	0	$1.45 \cdot 10^{-10}$	$2.02 \cdot 10^{-10}$	$9.09 \cdot 10^{-11}$
$\lambda = 0.5$	0	$1.27 \cdot 10^{-10}$	$1.52 \cdot 10^{-10}$	$1.00 \cdot 10^{-11}$

Table 3: MAE between different initialization, comparing each one with all values at $\frac{1}{n}$.

References

[1] Victor Lavrenko. Sink nodes in pagerank, 2014.