PageRank

CAIM Lab, Session 5

Oscar Mañas, Anselmo López

About the algorithm

Below we provide a proof that translates the matricial version of the algorithm seen in class to its alternative notation:

$p = G^T \cdot p$

$G = \lambda \cdot M + (1 - \lambda) \cdot \frac{1}{n} \cdot J$

$p = (\lambda \cdot M + (1 - \lambda) \cdot \frac{1}{n} \cdot J)^T \cdot p = (\lambda \cdot M^T + (1 - \lambda) \cdot \frac{1}{n} \cdot J^T) \cdot p = \lambda \cdot M^T \cdot p + \frac{(1-\lambda)}{n} \cdot J \cdot p =$
$= \lambda \cdot M^T \cdot p + (\frac{(1-\lambda)}{n}, ..., \frac{(1-\lambda)}{n})$

Then each element of *p* is computed as follows:

$p[i] = \lambda \cdot dot(M^T[i], p) + \frac{(1-\lambda)}{n}$

...and that's exactly each iteration of the algorithm.

The only thing left to mention is that we divide $w(i,j)$ by $out(j)$ because M^T needs to be a column stochastic matrix (each column summing to 1).

About the implementation

We take the skeleton program *PageRank.py* as a base for our own implementation of the algorithm that Larry Page and Sergey Brin developed in 1996. We decided to implement it in Python because we think it's a shorter and cleaner language than Java.

Since the efficiency of the implementation is important, we cannot use the matricial notation of the algorithm. If we have *n* vertices, matrices would have *n*n* elements, and this would take quadratic memory and time in the number of vertices. What's more, the number of edges of the graph we are dealing with is *O(n)* and not *O(n*n)*, so we are especially encouraged to write an implementation that takes linear memory and time in the number of vertices.

With that idea in mind, we organize our data structures in the following way. To store the airports, we have a list that stores objects of the class *Airport*, and then we have a dictionary that has IATA codes as keys and the indices of the airports in the previous list as values. That way, we can retrieve an airport given its IATA code in expected time *O(1)*.

Instead of having a list of edges corresponding to routes between airports, we store a list of incoming flights inside each *Airport* object. That way, the destination airport is implicit and we cut in half the space needed to store the edges. There's another reason to store edges this way: the algorithm needs to look for the flights that have a certain airport as a destination. So, our data structures are aware of their subsequent use.

To store the routes, we have a list that stores objects of the class *Edge*, and then we have a dictionary that has the IATA codes of the origin airports as keys and the indices of the routes in the previous list as values. As with the airports, this allows us to retrieve a route given the IATA code of its origin airport in expected time *O(1)*.

We decide to take convergence as the stopping condition, that is, when two P's at consecutive iterations are sufficiently close. In our implementation, we've defined "sufficiently close" with a precision of 15 decimal places, since we think this value gives a good tradeoff between accuracy and performance.

Another thing to take into account is airports without outgoing routes. This is important since for the algorithm to work properly, we need that the matrix is column stochastic. If there are airports with no outgoing routes, the matrix will have columns with all zeros, so the sum of those columns will be 0. To fix that, we create an outgoing route to a random destination airport for each airport that has no outgoing routes.

As suggested in the statement, after each iteration we checked that the sum of all weights was 1 (or at least close to 1, since there are some precision errors). We also checked that all columns of the implicit matrix added up to 1. But this second assertion failed for some of the columns. After spending hours looking for the reason, we discovered that in the *airports.txt* file there are 2 cases of 2 airports that have the same IATA code: BFT and ZYA. Since we use the IATA code of the airport as a unique identifier, our implementation obviously failed. We decided to remove the second airport in each case, and the program finally passed all the assertions.

About the results

We executed the final program several times changing the damping factor (λ, L in the program) to see its consequences. The most obvious difference is the value of each airport PageRank. Damping factor defines the chance of following the routes of the node. So, if we decrease the factor, we're telling the algorithm that users have a $(1-\lambda)$% chance of jumping to another random node. The result is that, while airports in high positions have less PageRank value, airports at the bottom receive the random jumps and increase their value.

The speed of convergence is also influenced by the changes. The number of iterations until reaching the stopping condition grows as the damping factor does. PageRank algorithm needs more iterations to achieve the convergence because the chance of following the links through nodes is higher and thus the weights vector needs more loops to become stable.

References

http://www.math.cornell.edu/~mec/Winter2009/RalucaRemus/Lecture3/lecture3.html