**Result Table:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Iteration | Contribution | A | B | C |
| 5 | 0.85 | 1.4965020996093743 | 0.7993014420572915 | 1.147901770833333 |
| 10 | 0.15 | 1.0697674444242142 | 0.9302325594201216 | 1.000000001922168 |
| 10 | 0.5 | 1.200000423855272 | 0.8000002119276056 | 1.0000003178914387 |
| 10 | 0.85 | 1.3857307330281197 | 0.7455188698656946 | 1.0656248014469072 |
| 10 | 1 | 1.7777776718139648 | 0.8888889948527018 | 1.3333333333333333 |
| 15 | 0.85 | 1.3370699140795597 | 0.7211662319879406 | 1.0291180730337501 |
| 20 | 0.85 | 1.3154720694847448 | 0.7103676179049304 | 1.0129198436948377 |

**Short Answer of findings:**

From the computation part, I found that PageRank calculation convergence is very efficient. From the paper, I also learn that the convergence is highly scalable too. This is good, because we could run less iterations to finish the computation, and it becomes easier to find a good number of iterations too.

Moreover, the contribution provides a control for the speed of convergence. By setting contribution, we can easily decide the “step” each iteration would move forward.

For the result part, by using PageRank we could quickly rank elements in a meaningful order efficiently. It avoids a lot of problem such as fraud links, meaningless links, etc.

It could be a good indicator of the characteristics of the elements and also a good indicator to predict many other properties.

In the case we studied as an example, by few over ten iterations and 15 seconds, we could quickly rank the four elements into an order of A-C-B-D, where D has no PageRank. The convergence is really quick and effective. By having this result, we not only know the relatively positive of the pages in this network, but also we can use it to predicts other properties, such as page traffics and the reputation of the pages in the professional field.

**Coding Part:**

lines = sc.textFile("s3://smokeeveryday/data420/Relation.txt")

lines.collect()

links = lines.map(lambda nodes: nodes.split(",")).groupByKey()

links.collect()

ranks = links.map(lambda nodes: (nodes[0], 1.0))

ranks.collect()

def computeContribs(nodes, rank):

num\_nodes = len(nodes)

for node in nodes:

yield (node, rank / num\_nodes)

from operator import \*

for iteration in range(20):

contribs = links.join(ranks).flatMap(lambda nodes:

computeContribs(nodes[1][0], nodes[1][1]))

ranks = contribs.reduceByKey(add).mapValues(lambda rank: rank \* 0.85 + 0.15)

ranks.collect()