Monte Carlo PageRank Algorithm Implemented in Apache Spark

Sam Campbell
University of Waterloo
sj2campb@uwaterloo.ca

***Abstract*— Abstract goes here. Abstract goes here. Abstract goes here. Abstract goes here.**

I. Introduction

Citation examples: [1], [2], [3], [4].

The PageRank algorithm has been widely used and researched since its introduction in 1998, due in part to its usefulness for ranking web pages at Google to improve their search platform. Its purpose, when used in the context of web pages, is to rank the importance of web pages based on the inter-page link structure. Pages with more links pointing to them get a higher rank. PageRank values can be used for various applications, like ordering search results, browsing, or estimating web site traffic.

The size of the entire web graph is very large and continues to grow, so for the PageRank algorithm to be effective, it should run as efficiently as possible. There is more than one way to compute PageRank. Two of them are discussed in this paper: a Power Iteration (PI) approach and a Monte Carlo (MC) approach. The PageRank algorithm that was presented in 1998 by L. Page, S. Brin, R. Motwani, and T. Winograd [3] uses the PI approach, which uses an iterative matrix-vector multiplication technique to calculate PageRank. A Monte Carlo approach was introduced by K.Avrachenkov et al. in 2007 [1]. You can imaging the MC approach as a series of random walks through the graph and PageRank for a node is relative to the number of times any walk goes through a node.

Much research has gone into improving the performance of PageRank computation, though most research appears to be on improving the performance of the PI approach. More recently, a few MC algorithms were introduced by Das Sarma et al. [2] to calculate PageRank on a distributed system. In their research, they prove theoretically that one of their algorithms runs in O(log(n/E)) number of rounds for directed or undirected graphs (like the web), and another runs in O(sqrt(log(n/E))) rounds for undirected graphs only, however, they did not have an implementation to show their results.

The entire web graph is very large, so even with an efficient algorithm, PageRank may not be able to run very quickly on a single system. The ClueWeb09 dataset contains the structure of the web from 2009. The dataset containing all unique URLs is 325GB uncompressed. One way to tackle the large dataset is to distribute the PageRank algorithm so that multiple computers are working on the calculation at once. This is where Apache Spark comes into play. It was built for fast cluster computing and large-scale data processing.

Ive implemented an MC PageRank algorithm described by Das Sarma et al. [2] in Spark. They introduce two algorithms: Basic-PageRank-Algorithm (BPRA) and Improved-PageRank-Algorithm (IPRA). However, only BPRA works for directed graphs like the web, so this paper will focus only on that moving forward. Ive also implemented a PageRank algorithm using the PI approach in Spark, and Ive compared the results. Ive specifically compared the convergence of the algorithms by looking at the PageRank of the 1st page, 10th page, 100th page, and 1000th page to show how the PageRank distribution differs between the two approaches after a number of iterations.

II. PageRank

A PageRank vector for a graph is defined as , where each element in represents the PageRank of a node in the graph.

A. PowerIteration PageRank

* Describe the random surfer model
* Brief Description of PageRank Power Iteration (PI) Method

B. Monte Carlo PageRank

The random surfer model used in the original PageRank computation can also be represented by the MC approach using random walks. In this sense, a single surfer can be represented by one random walk wij, starting at node i and ending at node j. There is a probability, ϵ, that the surfer will click away to a random page, so $(1 - \epsilon)$ is the probability that they will follow an outgoing link from their current page. These basic rules can be used to guide the random walk.

To describe one step in a random walk from node i, a random number would be generated. If the number is below ϵ, then the walk ends. Otherwise, the walk picks one of the outgoing links with equal probability, which would be 1/nO, where nO is the number of outgoing links from the current page.

During a random walk, each node must record that it has been visited. The total number of visits to node v is denoted by ζ_v. The total number of visits over all nodes of all walks is $\frac{nK}{\epsilon}$. The PageRank for a node v, denoted as π_v, is estimated by dividing the number of visits to a node by the total number of all visits. The Monte Carlo approach is naturally an approximation of the actual PageRank values, so an estimation of π_v is denoted by $\tilde{\pi}_v$. Therefore,

$$\tilde{\pi}_v = \frac{\zeta_v \epsilon}{nK} \tag{1}$$

There are a few more important notations involved in the algorithm. T_v^u is the number of random walks from v to u in a single round. B is a constant used in calculating the number of rounds that the algorithm should run for: $B \log n/\epsilon$. This algorithm is laid out by Das Sarma et al. in [2], and is summarized here in algorithm 1.

Algorithm 1 Basic-PageRank-Algorithm [2]

```
Input: Number of nodes $n$ and reset probability $\epsilon$
Output: PageRank of each node
[Each node $v$ starts $K$ walks. All walks occur in parallel until they terminate. Termination probability for each walk is $\epsilon$, so the expected walk length is $1/\epsilon$ ]
1:  Each node $v$ maintains a count variable '$couponCount_v$' corresponding to number of random walk coupons held by $v$. Initially, '$couponCount_v = K$' for starting K random walks.
2:  Each node $v$ also maintains a counter $\zeta_v$ for counting the number visits of random walks to it. Set $\zeta_v = K$
3:
4:  for round $i = 1, 2, ..., B \log n/\epsilon$ do
5:      Each node v holding at least one alive coupon (i.e., $couponCount_v \neq 0$) does the following in parallel:
6:      For every neighbour $u$ of $v$, set $T_u^v = 0$
7:      for j = 1,2,...,$couponCount_v$ do
8:          With probability $1 - \epsilon$, pick a uniformly random outgoing neighbour $u$
9:          $T_u^v := T_u^v + 1$
10:     end for
11:     Send the coupon counter number $T_u^v$ to the respective outgoing neighbours $u$.
12:     Each node $u$ computes: $\zeta_u = \zeta_u + \sum_{v \in N(u)} T_u^v$
13:     Update $couponCount_u = \sum_{v \in N(u)} T_u^v$
14: end for
15:
16: Each node $v$ outputs its PageRank as $\frac{\zeta_v \epsilon}{nK}$
```

III. Implementation

Both the Power Iteration and Monte Carlo algorithms were implemented in Apache Spark. Spark is a system designed for running large-scale distributed computations, and abstracts the low-level details pertaining to managing the complexities of distributed computing. Similar to Hadoop's MapReduce, it brings the computation to where the data is stored. For instance, a large graph could be split up and stored across many servers. To run a Spark program on a distributed dataset, the program is passed to each Spark node, and executed on the nodes portion of the dataset.

A. PowerIteration Implementation

Describe PI implementation here

B. Monte Carlo Implementation

To describe how Spark implements this algorithm, think of it in terms of a single random walk w_{ij} that goes from node i to node j. Only one step of each random walk will be carried out within one iteration. One iteration can be carried out using map-reduce-like functionality. Information will have to be passed from node i to node j, but since the graph is distributed, these nodes could be on separate computers. The way this can be handled in Spark is by outputting a key-value pair for each random walk after each iteration, so for a random walk w_ij, we can output $(j, 1)$. This shows that a random walk has moved to node j. To figure out where the random walk will go next after node j, we need to find the outgoing nodes from j. Spark infrastructure handles the complicated distributed computing components of this operation by ensuring that this output gets joined back up with the link structure of node j, which may or may not be on a different computer in the cluster. This computation will run most efficiently if the graph is distributed in a way that the fewest edges are broken when splitting the graph across Spark systems.

IV. Methods and Data

The PageRank implementations were run against a simple graph dataset: the Gnutella perr-to-peer network graph structure from 2002. This dataset is only 128KB, consisting of 6301 nodes. This is basically a toy dataset to test the convergence capabilities of the two PageRank algorithms quickly. Further work could expand upon this by running this on a much larger graph.

TODO: Explain the experiment: convergence.

V. Results and/or Discussion

VI. Conclusion

References

[1] R.R. Anderson and J.A. Parrish, "Optical properties of human skin," in *The Science of Photomedicine*, J.D. Regan and J.A. Parrish, Eds., N.Y., USA, 1982, pp. 147–194, Plenun Press.

[2] C. Antoniou, J. Lademann, S. Schanzer, H. Richter, W. Sterry, L. Zastrow, and S. Koch, "Do different ethnic groups need different sun protection?," *Skin Res. Technol.*, vol. 15, no. 3, pp. 323–9, 2009.

[3] G. Golub and C. Van Loan, *Matrix Computations*, John Hopkins University Press, Baltimore, MD, USA, 2nd edition, 1989.

[4] M. H. Ravnbak, *Objective determination of Fitzpatrick skin type*, Ph.D. thesis, University of Copenhagen, 2010.