Running PageRank with Kiji

Shashir Reddy

April 5, 2013

1 Introduction

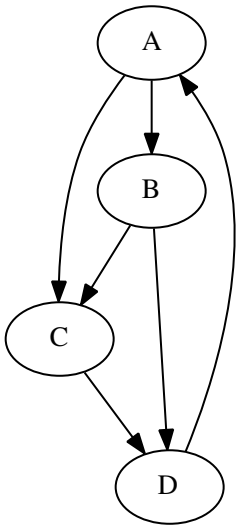
PageRank was developed at Stanford by Larry Page and Sergey Brin who went on to found Google and revolutionized technology. PageRank is a heuristic for the relevance of a web page and can be used to enhance results of web search queries. Since it is *the* classical distributed computing problem of the extant generation of search engines, we shall attempt to compute it using Kiji for every page in the English Wikipedia. Additionally, I am sure everyone is bored of word-count demonstrations on Hadoop – this should be a nice distraction.

All the code is available in the accompanying archive file and the parsed links data sets are available on Henry Haselgrove's Using the Wikipedia links dataset.

2 Background (hand-wavily peppered with probability-theoretic terms)

PageRank conceives of the web as a large directed graph where nodes represent web pages and edges represent one-way links. Suppose a random walker is dropped onto an initial page and mandated to follow links forever (let's say he follows any link on a page with equal probability). What is the probability that he will land on any other given page?

Figure 1: The web as a directed graph.



Formally, we have described a random process known as a Markov chain. The probability of following a link from one page to another can be described by a column-stochastic transition matrix M where each entry m_{ij} represents the probability of transitioning from page j to page i. In the case of Figure 1:

$$M = \begin{pmatrix} 0 & 0 & 0 & 1 \\ \frac{1}{2} & 0 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 1 & 0 \end{pmatrix} \tag{1}$$

Intuitively, applying the transition matrix on a stochastic vector $\pi = (\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4})$ an indefinite number of times produces the stationary probability distribution of the random walker's landing on each page. This is the power iteration method for discovering the fixed point of the transition matrix, i.e. we are seeking π^* such that $\pi^* = M\pi^*$. For the web graph in Figure 1.

$$\pi^* = \lim_{i \to \infty} M^i \pi = (0.\overline{36}, 0.\overline{18}, 0.\overline{09}, 0.\overline{36})^T \tag{2}$$

This probability distribution is interpretable as the rank of every page. However, in order for such a stationary distribution to exist, the web graph must be strongly connected, i.e. all states must be reachable from one another – this certainly is not true in reality. If we include an accessible page

with no outlinks, the rank of all the pages goes to 0. If we include an accessible strongly connected subgraph with no outlinks, then the rank becomes monopolized by that subgraph. This is commonly known as a "link farm" and has been used to exploit rank.

In order to combat illegitimate link farms, PageRank implements "random teleportation." At any time, the random walker is expected to give up surfing on his current path and to randomly teleport to another page with a non-zero probability σ (we will go with 15% for this project). This amounts to a [farm] taxation on all pages and egalitarian redistribution of rank. The iterative computation becomes $\pi' = (1 - \sigma)M\pi + \frac{\sigma}{n}\mathbf{1}$ where $\mathbf{1}$ is the vector of all 1s and n is the size of the graph.

3 Data source and Kiji setup

Wikipedia is a heavily curated and highly interpretable webgraph and its data dumps are easily accessible online. In this setup, I use data sets containing parsed descriptions of all the links in the English Wikipedia. These data sets are available on Henry Haselgrove's Using the Wikipedia links dataset (his parser for the raw Wikipedia data is also available there). He too computes PageRank and we may compare our results to his.

We load all our data into a WEBTABLE table in Kiji (using the layout file WEBTABLE_layout.json). There is only one locality group ("family" in HBase language) and one family *info*. Individual pages represent the entities in the table (the entity-id is the page id). The page title is given by the column *info:title*, the score by column *info:score*, each outlink's page id is provided with an incremental timestamp by column *info:links*, and the number of outlinks for a given page is given by column *info:numoutlinks*. For example:

entity-id = [123] :	info:title = Widgets
	info:score = 0.0012
	info:links = $\langle 134, 122 \rangle$
	info:numoutlinks = 2

Note: simply for the calculation of PageRank, it is more efficient to sequentially read a file straight from HDFS instead of using a Kiji/HBase table.

However, the entity-centric model enforced by Kiji/HBase tables would be useful to expand records with additional features such as anchor text, inlinks, other scores, et cetera. By using the Kiji framework for this project, we are contradictorily trying to be both realistic and slightly unpractical.

4 MapReduce computation

To compute the rank of a single page, we must iteratively sum the outlink-weighted ranks of all the pages pointing to it (corresponding to dotting a row in the matrix M with the vector π containing the current distribution of ranks). This can be accomplished with the gatherer (a type of mapper) RankRedistributor and the reducer RankCollector.

Without random teleportation (since it is simpler to read), RankRedistributor would take each record in the WEBTABLE and proportionately distributes the rank to each of the outlinks (with each link as the key and equally divided rank score as the value).

$$\begin{gathered}\text{RankRedistributor}: \\ [\text{page}, \text{score}, \langle \text{link1, link2, ...} \rangle, \text{numoutlinks}] \xrightarrow{emits} [\text{link1}, \text{score/numoutlinks}] \\ [\text{link2}, \text{score/numoutlinks}] \\ [\text{link3}, \text{score/numoutlinks}] \\ \dots\end{gathered} \tag{3}$$

RankCollector would receive each link as the key and an iterable list of rank scores that were distributed to it by the RankRedistributor. This list is summed and the total score is written to the *info:score* column of each entity back in the WEBTABLE.

$$\text{RankCollector} : [\text{page}, \langle \text{score1, score2, ...} \rangle] \to \sum_i \text{score}_i \tag{4}$$

In order to incorporate random teleportation, we tax the rank scores emitted by RankRedistributor by a certain amount – say 15%. The RankCollector then adds the social security check of $0.15/n$ where n is the size of the web. It's very socialistic.

There are some caveats to note in this implementation:

Firstly, pages with no inlinks would never be parsed by the RankCollector. Therefore, their social security check would not be collected and their rank would not correctly trickle back into the economy. To get around this, RankRedistributor also emits the value $[page, 0]$, thereby forcing the RankCollector to consider every page, including those with no inlinks.

Secondly, pages with no outlinks can not redistribute their score to the rest of the web with the MapReduce framework (there are just too many pages to emit). Instead, a heuristically determined flat rank (largely negligible) is added back by the RankCollector on behalf of all the dead node pages.

5 Running the works and results

The above process was run on a 6-node cluster with 128 map tasks and nearly 100 reduce tasks. I went with such a large number of reducers because pages which have a large number ($>$ a hundred thousand) inlinks may force their RankCollectors to block ("convoy"-like effect) the reducer slots which can otherwise compute rank for less popular pages.

Every iteration took an average of 11 minutes and within about 7 iterations, the order of the top 10 pages essentially stabilized. I rank a total of 13 iterations; the ranks of the top 15 pages are displayed below.

Title	iteration = 13
United States	0.0022550933
2007	0.0014943096
2008	0.0014371843
United Kingdom	0.0011267669
Geographic coordinate system	0.0009519762
2006	0.0009477666
Wiktionary	0.0008980834
Wikimedia Commons	0.0008972345
France	0.0007945098
Canada	0.0006747068
2005	0.000660031
Germany	0.0006579494
English language	0.0006364806
England	0.0006191937
World War II	0.0006090324

This is quite comparable to Haselgrove's results also (I am not sure how many iterations he performed).

6 Concluding thoughts

This experiment has been mostly a learning experience for me as a novice hacker on Kiji (and as a novice hacker period!). My interactions with the underlying Hadoop and HBase were largely limited and I was able to rely on the abstractions in Kiji. I faced some difficulty in determining the number of regions to split my WEBTABLE into (this affects the number of mapper threads spawned) – something most HBase users encounter presumably early in their careers. My bulk imports were expectedly very slow since I initially used individual writer puts instead of HFiles. I am still fiddling with the parameters to more quickly compute every iteration. I must reiterate that for the purpose of fast PageRank calculation, sequentially reading link data from HDFS flat files would be significantly more efficient than using Kiji tables, but the entity-centricity provided by Kiji tables allows for expansion of columns for additional features such as anchor text, inlinks, other scores, et cetera