|  |
| --- |
| CSC 570 - Big Data Analytics |
| Analyzing the ACM Citation Network |
| Fall 2016 –Final Project Option A |
|  |
| **Justin Orr** |
| **11/26/2016** |

|  |
| --- |
|  |

Tables of Contents

Analyzing the ACM Citation Network ……………………………………………………………………………….page 2

Building the Citation Network using Spark ……………………………………………………………………….page 2

Performing Graph Analytics on the ACM Citation Network ………………………………………………page 3

Visualizing the in-degree distribution

Performing Graph Analytics on the ACM Citation Network ………………………………………………page 5

Implementing the Weighted Page Rank Algorithm

Performing Analytics on the ACM Citation Network …………………………………………………………page 8

Results and Conclusion

References ……………………………………………………………………………………………………………………….page 9

Analyzing the ACM Citation Network

The ACM Citation Network data set is a collection of citation data which can be used for a clustering network and side information, studying influence in the citation network, finding the most influential papers, topic modeling, and other research guided endeavors. The following report analyzes the ACM-Citation-Network V8, which consists of 2,381,688 papers and 10,476,564 citation relationships. The requirements of building this network involves multiple technologies, including Scala, Apache Spark, and Apache Hive; then performing graph analytics such as in-degree distribution and weighted page ranks. Once this data has been filtered and configured, the final steps include analyzing the results to determine which ten papers were most influential.

Building the Citation Network using Spark

In order to analyze the citation network, the first step is to extract the citation graph from our data set and produce a new set of the form:

**Paper-index 1 Paper-index 2**

where Paper-index 1 cites Paper-index 2. To do this in Spark, a few key configurations must be added to the solution to read in the format of the citation file. The citation file has a specialized format to distinguish each paper from another. Each paper’s ‘block’ is distinguished with two characters (‘#\*’) to mark the start of a new paper. The general format is as follows:

**#\* Paper Title**

**#@ Authors**

**#t Year**

**#c Publication Venue**

**#index <id of this paper>**

**#% ids of references of this paper (can be multiple lines for each reference)**

**#! Abstract**

Upon altering the configuration, the first necessary steps are reading in the file, filtering out any empty blocks of information, and mapping the blocks to a class that labels each extracted element:

**case class Paper(paper\_id:String, paper\_title:String, paper\_references:String)**

**val inputRDD = sc.newAPIHadoopFile(args(0), classOf[TextInputFormat],**

**classOf[LongWritable],classOf[Text], config)**

**val inputRDDMapFilter = inputRDD.map{case(key,value)=>value.toString}**

**.filter(value=>value.length!=0)**

**val paperData = inputRDDMapFilter.map{case(chunk)=>**

**Paper(getIndexKey(chunk), chunk.split('#')(0).trim, getReferences(chunk))}**

The process of doing so requires two special functions, getIndexKey() and getReferences(), designed to remove the ‘#index’ from the ‘paper\_id’ and each instance of ‘#%’ in the ‘paper\_references’, respectively. This information is then stored in an Apache Hive View called ‘papersView’ for later reference:

**val papersView = paperData.toDF.createOrReplaceTempView("papersView")**

Now that the information has been parsed, formatted, and stored, all of the papers that cite another need to be retrieved. This is done by querying the ‘papersView’ for all papers that cite at least one paper:

**SELECT paper\_id, paper\_references FROM papersView WHERE NOT paper\_references='Null'"**

This returns a dataset of the format (paper\_id, paper\_references). In order to build a proper citation graph, each paper needs to be in a one-to-one key-value pair of (paper\_id, paper\_reference). To do so, the final step is to flat map the values such that each value is an element of the comma delimited string ‘paper\_references’. Again, these values must be stored in a class that labels our extracted information:

**case class PaperReference(paper\_id:String, paper\_reference:String)**

**val flatData = pairData.rdd.map(row => (row(0).toString,row(1).toString))**

**.flatMapValues(refs=>refs.split(','))**

**.map(pair=>PaperReference(pair.\_1,pair.\_2))**

At this point, the ‘flatData’ dataset is the necessary citation graph sought after. Now, this citation graph can be used to perform different analytical calculations.

Performing Graph Analytics on the ACM Citation Network

Visualizing the in-degree distribution

The first of two calculations run on the citation graph is calculating the in-degree distribution of the citation network. The in-degree of a paper is the total number of other papers which link to, or “cite”, the paper. The in-degree distribution p(k) is the fraction of papers in the network with a certain in-degree. To determine this, the use of a Spark library called GraphX helps to determine the total number of nodes (papers) and the total number of in-links to each node (other papers that cite it).

The GraphX library allows for the creation of a graph given a list of edges. Edges, in this case, are the links from one paper to another. This happens to be exactly what the citation graph is; a list of each paper and a paper it references. However, there is one problem with using GraphX. It requires that each edge be of a ‘long’ datatype. Since the paper ID’s are not numbers but unique identifiers containing both numbers and letters, they cannot be directly converted to a ‘long’ value representation. They must instead be converted to a unique, but repeatable ‘long’ value. In order to do so, a Scala library call MurmurHash3 must be utilized. A MurmurHash is a well-distributed, non-cryptographic hash function that is used for hash-based lookups. The Scala-based MurmurHash3 has a method ‘stringHash(string)’ which returns the integer hash code value of the given string. After this conversion, the hash data is used to create the graph:

**val hashedData = flatData.map{case PaperReference(index, reference)=>**

**(MurmurHash3.stringHash(index).toLong,**

**MurmurHash3.stringHash(reference).toLong)}**

**val graph = Graph.fromEdgeTuples(hashedData, null)**

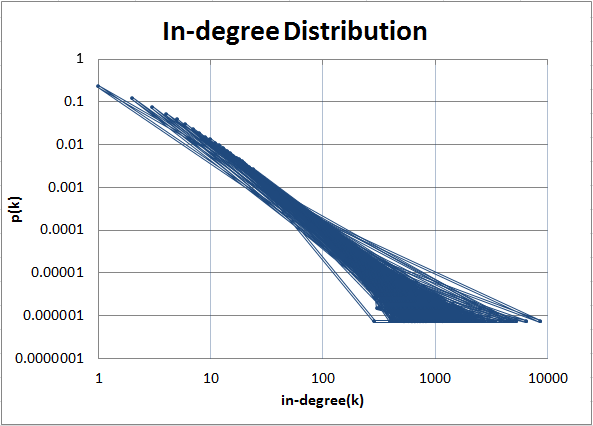
The next step is to use ‘graph’ to calculate the in-degree distribution. As stated above, calculating the distribution, p(k), will require the number of papers with ‘k’ in-links and the total number of papers ’n’: **P(k) = nk/n**. Since ‘n’ and the list of all in-degrees are default members of ‘graph’, only the number of times each in-degree appears will need to be calculated. This is achieved by mapping each in-degree to a key-value pair of (in-degree, 1). This represents the value of each individual in-degree’s occurrence. This dataset is then combined on the key and the associated values are summed, resulting in the total number of times each in-degree occurred (nk). The final step is to divide each of the sums by the total number of nodes (n) that exist in the graph and the in-degree distribution of each node will remain:

**val indegreeSums = graph.inDegrees.map{case(node,indegree)=>(indegree, 1)}**

**.reduceByKey((v1,v2)=>v1+v2)**

**val indegreeDist = indegreeSums.map{case(indegree,sum)**

**=>(indegree,sum.toDouble/vertCount.toDouble)}**



Performing Graph Analytics on the ACM Citation Network

Implementing the Weighted Page Rank Algorithm

The final calculation that needs to be run on the citation graph is the implementation of the weighted page rank algorithm. This will be used to find the top ten most influential papers in the network. This algorithm measures the importance of a page by the importance of the other pages that are linked to it. It is an extension of the standard page rank algorithm, which takes into account the importance of in-links and out-links of a page and distributes rank scores based on the popularity of a page. The weighted page rank PR(u) of a paper ‘u’ is calculated by:

**PR(u) = + d \***

Where ‘d’ is a constant value often set at .85, ‘N’ is the total number of papers in the citation network, ‘in(u)’ is the set of all nodes which link to node u, ‘Win’ is the in-weight of link(v,u), and ‘Wout’ is the out-weight of link(v,u). ‘Win’ and ‘Wout’ are calculated by:

**= =**

To begin, it is first necessary to calculate the in-links and out-links for each paper. The in-links can be calculated by using the citation graph from earlier and mapping each Paper Reference’s ‘paper\_reference’ value to the value 1. This is much like how the in-degree sums were calculated for the in-degree distribution. This key-value pair will then be mapped to a special class which labels the data:

**case class weightIn(wIn\_id:String, \_wIn:Int)**

**val wIn = flatData.map{case PaperReference(id,ref)=>(ref,1)}**

**.reduceByKey((v1,v2)=>v1+v2)**

**.map{case(x,y)=>weightIn(x,y)}**

Exactly the same thing is done to calculate the out-links, only this time each Paper Reference’s ‘paper\_id’ value is being mapped instead:

**case class weightOut(wOut\_id:String, \_wOut:Int)**

**val wOut = flatData.map{case PaperReference(id,ref)=>(id,1)}**

**.reduceByKey((v1,v2)=>v1+v2)**

**.map{case(x,y)=>weightOut(x,y)}**

Following these calculations, a dataset can be created by joining the citation graph with both the newly created ‘wIn’ and ‘wOut‘ data against the ‘paper\_reference’ value. This places what will eventually be the numerators of the ‘Win’ and ‘Wout’ calculations into an easily join-able dataset:

**val topWeightsJoin = flatData.toDF.as("links")**

**.join(wIn.toDF.as("win"), $"links.paper\_reference"===$"win.wIn\_id")**

**.join(wOut.toDF.as("wout"), $"links.paper\_reference"===$"wout.wOut\_id")**

**.select($"paper\_id", $"paper\_reference", $"\_wIn", $"\_wOut")**

Now that the numerators are known and stored, the denominators must be found so that an in-weight and out-weight can be calculated for each link. To calculate the denominators the ‘topWeightsJoin’ dataset can be used. This dataset is currently in the state:

**Paper\_Id Paper\_Reference Win Numerator(\_wIn) Wout Numerator(\_wOut)**

The denominator for In-Weight is shown to be the summation of in-links to a given node (or paper). That means if ‘topWeightsJoin’ is reduced by the key ‘paper\_id’ and all values are summed together, then the denominator will be the total of that summation. The same applies for the denominator for Out-Weight:

**val botIn = topWeightsJoin.rdd.map(row=>(row.getAs[String]("paper\_id"), row.getAs[Int]("\_wIn")))**

**.reduceByKey((v1,v2)=>v1+v2).map{case(x,y)=>weightIn(x,y)}**

**val botOut=topWeightsJoin.rdd.map(row=>(row.getAs[String]("paper\_id"), row.getAs[Int]("\_wOut")))**

**.reduceByKey((v1,v2)=>v1+v2).map{case(x,y)=>weightOut(x,y)}**

Now that the denominators are known, these values can be joined with the ‘topWeightsJoin’ dataset against the ‘paper\_id’ value to produce each links weight factor:

**val weightCalcs= topWeightsJoin.as("tw").join(botIn.toDF.as("bIn"), $"tw.paper\_id"===$"bIn.wIn\_id")**

**.join(botOut.toDF.as("bOut"), $"tw.paper\_id"===$"bOut.wOut\_id")**

**.select($"paper\_id", $"paper\_reference",**

**(($"tw.\_wIn"/$"bIn.\_wIn")\*($"tw.\_wOut"/$"bOut.\_wOut")).alias("weight"))**

All of the setup work to calculate the page ranks is now complete. A dataset exists that contains all of the page links and the associated weight factors.

The next step is to map the ‘weightCalcs’ dataset into a format such that the key can be grouped against. In this situation, mapping the key to be ‘paper\_id’ and value to be a tuple of ‘paper\_reference’ and ‘weight’ is the solution. Grouping this data set by key and persisting the outcome will allow for the upcoming iterations to not require the lazy evaluation to make this calculation over again. This step also requires the setup of the initial page rank for each node, which comes out to be 1/N (our total node count):

**val weights = weightCalcs.rdd.map(row=>(row.getAs[String]("paper\_id"),**

**(row.getAs[String]("paper\_reference"),row.getAs[Double]("weight")) ))**

**val links = weights.groupByKey().persist()**

**var ranks = links.map{case (id,links)=>(id, 1/vertCount.toDouble)}**

Since iterations over this dataset are required in order to allow the weights to differentiate the page ranks, this example of the weighted page rank algorithm will use ten iterations over the initial page ranks. During each of these iterations, the calculated weight value will be applied against the previous iterations paper rank. As shown above, each page rank calculation will require the summation of each in-link’s page rank multiplied by that links weight value. This summation, multiplied by ‘d’ and added to results in the new page rank for a given paper:

**def getWeightedRanks(links:Iterable[(String,Double)], rank:Double)**

**:Iterable[(String,Double)]=**

**{**

**for(l <-links) yield (l.\_1, l.\_2 \* rank);**

**}**

**for(x <- 1 to 10) {**

**val contribs = links.join(ranks).flatMap{case(id,(links,rank))=>**

**getWeightedRanks(links, rank)}**

**ranks = contribs.reduceByKey((c1,c2)=>c1+c2)**

**.map{case(id,sum)=>(id, sum \* .85 + .15/vertCount.toDouble)}**

**}**

The remaining two steps are quite simple. Step one is to select the top ten papers from ‘ranks’, which now contains all paper’s page ranks after ten iterations. All that need be done is sort the ‘ranks’ data set in descending order by ‘page\_rank’ value and select the top ten ‘paper\_id’ and ‘page\_rank’ pairs. The requested format of the solution is:

**Paper Title Number of Citations (In-Links) Page Rank**

Therefore, the dataset of top ten papers has to be joined with two earlier datasets. The first dataset is ‘paperData’, the dataset that all of the citation network was initially read into and stored with the associated class Paper. This dataset contains the ‘paper\_id’ and ‘paper\_title’ data required in order to retrieve the titles. The second dataset is ‘wIn’, the data set which contained each ‘paper\_id’ with its total number of in-links. Joining both of these data sets with ‘ranks’ against the ‘paper\_id’ allows for the selection of ‘paper\_title’, ‘number\_inlinks’, and ‘page\_rank’ of each of the top ten papers:

**val answer = topTen.as("tt").join(paperData.toDF.as("papers"), $"tt.\_1"===$"papers.paper\_id")**

**.join(wIn.toDF.as("inlinks"), $"papers.paper\_id"===$"inlinks.wIn\_id")**

**.select($"papers.paper\_title", $"inlinks.\_wIn", $"tt.\_2")**

**.sort($"tt.\_2".desc)**

Performing Analytics on the ACM Citation Network

Results and Conclusion

The final algorithm results in the list of the top ten most influential papers in the entirety of the ACM Citation Network dataset:

Singularity Theory and Phantom Edges in Scale Space 15 5.547660800419622E-5

A method for obtaining digital signatures and public-key cryptosystems 2223 3.560501257657747E-5

Chunking in Soar: The Anatomy of a General Learning Mechanism 86 2.931864129156816E-5

Authenticating Edges Produced by Zero-Crossing Algorithms 26 2.634733324670729E-5

Explanation-Based Generalization: A Unifying View 223 2.454552640196833E-5

Graph-Based Algorithms for Boolean Function Manipulation 2016 2.281473378344817E-5

The complexity of theorem-proving procedures 837 1.959916725929189E-5

Learning state space trajectories in recurrent neural networks 23 1.647007115751061E-5

Data clustering: a review 1701 1.506344506504665E-5

The Anatomy of the Grid: Enabling Scalable Virtual Organizations 1348 1.503048599101852E-5

One thing that stands out in the solution set is that almost half of the papers do not have a significant number of in-links (<100). One could stand to reason that the few papers that cited these individual papers went on to be quite influential themselves, thus helping elevate their importance. It’s also significant that the other half of the solution set has a substantial number of in-links (>1200), signifying how influential they have been to a broad spectrum of papers. Most likely making the top ten through sheer “brute force”.

Calculating the weighted page rank and in-degree distribution are just two of many possible uses for this massive dataset which consists of 2,381,688 papers and 10,476,564 citation relationships. Doing so would be extremely difficult without Big Data Analysis and technologies such as Hadoop, Scala, Apache Spark, and Apache Hive. It also goes without saying the research performed by engineers and mathematicians, such as Xing and Ghorbani[2], drastically helps to propel these technologies further ahead.

References

[1] <https://aminer.org/billboard/citation>, Yutao. "AMiner." AMiner. Web. 28 Nov. 2016.

[2] W. Xing and A. Ghorbani, "Weighted PageRank algorithm," Communication Networks and Services Research, 2004. Proceedings. Second Annual Conference on, 2004, pp. 305-314.

[3] <http://www.scala-lang.org/api/2.12.0-M4/scala/util/hashing/MurmurHash3>, Scala Standard Library 2.12.0-M4 - Scala.util.hashing.MurmurHash3. Web. 28 Nov. 2016.