May 12th, 2018

CS417: Data Mining

Wassim Gharbi, Lafayette College

Calculating PageRank & TopicRank

Introduction

This assignment aims to use the Hadoop/MapReduce framework to analyze and rank pages from a large dataset comprising of Wikipedia categories and articles from the " Wikipedia Network of Top Categories" dataset, a publicly available dataset assembled by the Stanford Network Analysis Project. The three-part assignments aims to leverage the properties of the MapReduce framework in order to automate the processing of a large graph of nodes obtained from crawling the Wikipedia website (most importantly hyperlinks between pages, page names and categories) in the order of 1.79 million nodes and 28.5 million links. In the first part of the

assignment we analyze the graph of nodes looking for dead-ends (which would be a problem for PageRank otherwise), on the second part we compute the PageRank of every node in the graph and sort the nodes by their obtained ranking. Finally, we compute the topic-specific PageRank (TopicRank) for our chose topic: **`Electrical_engineering`**.

Dataset Overview

The " Wikipedia Network of Top Categories" dataset is composed of a three large files spanning nodes, page names and page categories. The nodes file **`wiki-topcats.txt`** comprises source and destination nodes on every line, describing the inner citation graph of Wikipedia pages, we will use this file as the core input for our PageRank and TopicRank algorithms. The second file **`wiki-topcats-page-names.txt`** provides the original page names corresponding to every page ID. We use this file to label our results and get a clearer view of the relationships between pages (especially for TopicRank). The last file, **`wiki-topcats-categories.txt`** contains multiple top Wikipedia categories and a list of page IDs that belong to each of them. We use this file to obtain a list of pages belonging to the category of our choosing (**`Electrical_engineering`**) for the TopicRank algorithm.

Workflow and Debugging

One of the problems we commonly encounter with data mining in general and MapReduce in particular is the issue of slow processing time which causes debugging to become

a tedious task. For the purpose of faster development, we used two smaller datasets to test the correctness and soundness of the algorithms before processing the entire Wikipedia dataset.

The first dataset we used is a very minimal example graph composed of 4 nodes (A, B, C and D) and with an adjacency list small enough that it allows us to manually compute the PageRank over 10 iterations (comparable to a textbook exercise). Through this graph and the known PageRank we were able to minimize runtime (to the order of seconds) and debug the program extremely easily. Since this minimal dataset had the same structure as the Wikipedia one, it was a trivial task to switch to computing PageRanks for the full dataset.

The second dataset we used is a smaller split of nodes (first 1000) manually curated from the Wikipedia dataset. We named files in this smaller split **`wiki-topcatsXXXX-excerpt.txt`** and we created a boolean flag inside our program that allows us to toggle between using the full dataset and the excerpt dataset that we created. This dataset was particularly useful for DeadEnds algorithm since it allowed us to manually "cause" dead-ends.

Once the algorithm was validated, we ran it on the full dataset (**`wiki-topcats.txt`**) on an Amazon Web Services EMR cluster in order to process the full ~28.5m orders.

DeadEnds Algorithm Architecture

The DeadEnds MapReduce algorithm (provided in **`DeadEnds.java`**) was particularly easy to devise given that its architecture is very similar to the Frequent Items Pairs algorithm. To solve the problem of finding dead ends on the graph (nodes that have no out-links), we reduced the problem to counting out-links for each node of the graph (its out-degree) and looking for nodes that have a 0 as their out-degree. This is a single-job algorithm that simply emits a one for every origin node and a zero for every destination node then sums up the ones inside the reducer to obtain the final out-degree of the node. Within the reducer, only nodes with a null out-degree would be emitting to the final output (these are the dead-end).

The structure of the algorithm is detailed in the following table:

Job	Step	Goal	Mappers/Reducers	Functionality	Output Location
1	deadends	Obtain a list of nodes with a null out-degree (no out-links and therefore are dead-ends)	OutLinkMapper	Emits a key-value pair where the key is the node ID and the value is 1 for a source node or 0 for a destination node	./deadends-output
			NullOutDegreeReducer	For each node ID, sums up the received values to determine the out-degree and only outputs nodes with a null out-degree.	

It is worth noting that we have also provided an **`OutDegreeReducer`** class that we used for debugging which instead of outputting nodes that have a null out-degree, would output all nodes and their corresponding out-degree.

DeadEnds Algorithm Results

Through this algorithm we discovered that there are no dead-ends in the Wikipedia Dataset, meaning that every page has at least one outgoing hyperlink to another page. This is expected given the complex structure of the Wikipedia website. The result is provided as an empty list inside the **`./deadends-output`** folder. However, to confirm the correctness of the algorithm, we also ran it on the excerpted dataset we use for debugging (first 1000 nodes), and the program identified multiple dead-ends as expected. The fact that no dead-ends were spotted in the dataset means that our PageRank algorithm will not have to deal with this problem (especially that we are not implementing random teleports).

PageRank Algorithm Architecture

In order to devise the PageRank MapReduce algorithm (provided in **`PageRank.java`**) we divided the process into four majors steps: creating the initial PageRanks matrix, a power iteration step that will allow us to converge the PageRanks over **`k`** iterations (with **`k`** being 10 for this assignment), joining the resulting page IDs and PageRanks to their corresponding page names and finally sorting and outputting the sorted results list.

The algorithm was straightforward, however the implementation and debugging were challenging. To compute the initial matrix, we simply go through the edges and emit every edge from the mapper (in the form of a tuple with source and destination node IDs). In the reducer, we build an adjacency list for each node and at the same time we assign an initial page rank of **`1/n`** (with **`n`** being the total number of nodes, **`1791489`** in this case).

An implementation problem we encountered was how to represent a node inside MapReduce (and more importantly, how to serialize it and deserialize it to pass the node's adjacency list and PageRank between jobs). Luckily, we stumbled across an interesting built-in concept of **Custom Writables**. This allowed us to implement a serializable Hadoop **`Writable`** type that can easily be read and written in intermediate files while maintaining its structure as a Java class by simply extending the Hadoop **`Writable`** class and overriding its **`read`** and **`write`** methods. This did however come with the restriction of using binary intermediate files instead of text files (so that Hadoop can easily encode and decode our custom type into bytes as

opposed to strings), which meant that our intermediate files were illegible but this was a compromise we were willing to make given that it made our code cleaner and we didn't have to deal with manually encoding values into strings and parsing them back on the next step. Given this custom **`Node`** type that we implemented, all we had to do was to build the adjacency list in the reducer given the origin node and the list of destination nodes (passed on from the mappers).

The iteration step of the PageRank algorithm was a bit trickier but relied on a simple concept: In the mapper, we would emit the fraction of PageRank weight that each destination node would incur (which is the source node's page rank divided by its out-degree) and in the reducer we would sum up these weights giving us the new PageRank for the node. However, the issue with this algorithm is that we have to preserve the adjacency list as well for the next iteration, which poses a problem of output types (since our mapper would output both adjacency lists and PageRanks). Luckily, our custom **`Node`** type came to the rescue and we used it as a data structure similar to "**`Union`**"s in C (where, during serialization we would use the first bit to encode a boolean indicating whether the object is a PageRank or a Node then based on that we encode the different following values). We then repeat this iterative step **`n`** times and proceed to the next steps.

In the last two steps, we added the page names to the resulting list of page IDs and PageRanks (using the **`MultipleInputs`** feature of MapReduce, which gives us the ability to read multiple files through different reducers and output to the same reducer), then we sorted the output by PageRank using the same key/value flipping trick we used in the last project.

The structure of the algorithm is detailed in the following table:

Job	Step	Goal	Mappers/Reducers	Functionality	Output Location
1	initialize	Builds **`Node`** objects (comprising adjacency lists and initial PageRank of 1/number of pages)	IntializationMapper	Emits a key-value pair where the key is the source node and the value the destination node	./intermediate_0
			IntializationReducer	For each node and its list of destination nodes, creates a **`Node`** object that tracks the adjacency list and an initial PageRank	
2 . . . 11	iteration_0 . . . iteration_9	Calculates PageRank through an iterative process over 10 iterations	PageRankIterMapper	Emits to each destination node a fraction of the PageRank which is the source node's PageRank divided by its out-degree and emits the **`Node`** object itself to the source node (to preserve the adjacency list for the next steps)	./intermediate_1 . . . ./intermediate_10
			PageRankIterReducer	If a **`Node`** object is received, stores it locally, otherwise it sums up the PageRank fractions to become the new PageRank and once the **`Node`** object is received, it updates its PageRank and re-emits it.	
12	naming	Adds names to the results list	PageRanksMapper	Reads the list of page ID and their page ranks from the last iteration and outputs a tuple with the page ID as a key	./intermediate_11
			NamesMapper	Reads the list of page IDs and their names from the input file and outputs a tuple with the Page ID as a key	

			NamesReducer	Joins the PageRanks, page IDs and page names based on the page ID (natural join).	
13	sorting	Sorts the result list in descending order of PageRank	SortMapper	Inverses the order of keys and values in order to allow the data to be sorted by value instead of by key (combined with a custom comparator class that sorts in descending order instead of the default ascending order)	./pagerank-output
			SortReducer	Re-inverses the order of keys and values in order to get back the original structure of the data after sorting by value.	

PageRank Algorithm Results

We ran this algorithm on the entire dataset. The results are provided in the **`pagerank-output`** folder detailing the page IDs, names and PageRank for every node in the dataset sorted by PageRank. The first few entries are provided below:

```
279122 United States    0.005311657
541013 United Kingdom   0.002480993
987583 France   0.001986177
121347 World War II 0.001738354
896828 Germany  0.001665337
230038 Canada   0.001564982
1055792 English language    0.001524020
98332 Italy 0.001307694
1118496 India   0.001238352
610154 Australia    0.001231448
362517 Japan    0.000987314
```

Figure. Highest ranking pages in the Wikipedia dataset

To analyze whether the first node (United States) is part of a Spider Trap, we implemented a second version of our **PageRank** algorithm with a dampening factor. This will effectively tax pages that are part of a Spider Trap. We therefore expected the United State's PageRank to get lower if it were part of a Spider Trap. With a taxation of 15% ($\beta = 0.85$), we obtained the following results which can be found in the **`pagerank-output-with-taxation`** folder:

```
 1 279122 United States    0.083503660
 2 541013 United Kingdom   0.031938300
 3 987583 France   0.028458531
 4 230038 Canada   0.023041588
 5 121347 World War II 0.022645839
 6 896828 Germany  0.021904383
 7 1055792 English language    0.020103497
 8 610154 Australia    0.017678340
 9 98332 Italy 0.017604878
10 1118496 India   0.015439729
11 362517 Japan    0.015271660
```

Figure. Highest ranking pages in the Wikipedia dataset with a 15% tax

Since the United States' PageRank did not decrease, we are led to believe that this node is not part of a Spider Trap.

TopicRank Algorithm Architecture

The TopicRank algorithm was very similar in architecture to the PageRank algorithm. The minor change we introduced was in the **`PageRankIterReducer`** class which uses another formula to take into account whether a page is part of the given topic or not. The portion of the code that we changed is displayed below:

```
Node m = null;
double pageRank = 0.0;
for (Node pageRankOrNode:pageRanksOrNode) {
  if (pageRankOrNode.isNode()) {
    m = Node.clone(pageRankOrNode);
  } else {
    if (TOPIC.contains(nodeId.get())) {
      pageRank += BETA * pageRankOrNode.pageRank + (1 - BETA) / TOPIC.size();
    } else {
      pageRank += BETA * pageRankOrNode.pageRank;
    }
  }
}

m.setPageRank(pageRank);
context.write(nodeId, m);

```

Figure. Implementation of the formula for computing the TopicRank

In essence, the node's new PageRank is now computed as $\sum_{i}^{j} \beta \frac{r_i}{d_i} + (1-\beta)/|S|$ (with S being the nodes belonging to the same topic) if the page belongs to the topic, otherwise we use the same formula as the regular PageRank algorithm.

All other 4 major steps remained from the PageRank algorithm.

TopicRank Algorithm Results

We ran the TopicRank Algorithm on the entire dataset with our chosen topic of **`Electrical_engineering`** in order to evaluate its effectiveness. The results we obtained are provided in the **`topicrank-output`** folder. The first few entries are displayed below:

```
252920 Electrical engineering   1.188082116
131480 Electric motor   0.978332558
252907 Direct current   0.615701132
252906 Alternating current  0.614218888
255709 Printed circuit board    0.277615761
253457 Electric power transmission  0.249189868
252702 Microelectromechanical systems   0.134678031
252881 Electrical network   0.108285166
252878 Short circuit    0.095569600
252891 Electronic engineering   0.094420281
252785 Control engineering  0.090224860
252857 High voltage 0.089269027
252883 Voltage  0.086444019
279122 United States    0.083201029
253432 Three-phase electric power   0.080327050
253751 Electrical wiring    0.066570887
252843 Electric current 0.064636278
253686 Thomas Edison    0.063170244
```

Figure. Highest ranking pages in the Wikipedia dataset in the **Electrical engineering** topic

Given that almost all of the top results are directly or indirectly related to the field of Electrical Engineering (including concepts, components, notable people, etc.), we can safely say

that the algorithm is performing well. The only anomaly we can observe is the “United States” page. However, given that the page ranks high in the topic-agnostic PageRank measure, it is not unusual to see it ranking in the top 20 pages of Electrical engineering pages.