Churn

John Li, Albert Ge, Timothy Chou, Kevin Chang

Rankmaniac Report

1 Overview

During this RankManiac project, our group "churn" implemented PageRank on the given network via Map Reduce through Hadoop on Amazon AWS Clusters. Two sample datasets, "GNPn100p05" and "EmailEnron", were distributed to us for the purpose of local testing. Our Rankmaniac score was determined by our PageRank's elapsed time and the correctness on a larger, hidden dataset on these Amazon AWS clusters. This report's purpose is to illustrate the process for our implementation of PageRank.

2 Initial Framework

Development

The data that was given to us was in the form:

NodeId:[NODEID] [CURRENT_RANK],[PREVIOUS_RANK],[ADJACENCY_LIST]

After a basic understanding of the structure of the project, our initial framework of MapReduce was as follows:

- PAGERANK_MAP:
  The input of this mapping will either be in the above format, or the output of PROCESS_REDUCE, which prepends a [ITERATION] value in front of the current values. This mapping function will take in the key/value pair, increment [ITERATION] by 1, and map the key/value pairs to be either:

  [NODEID],[FLAG],[CURRENT_RANK],[ITERATION] [ADJACENCY_LIST]

  or

  [NEIGHBORID],[FLAG],[CURRENT_RANK],[ITERATION] [NEIGHBOR_RANK]

  NOTE: [FLAG] represents a string with the values : "rank" or "list ". This is specify which type of key/value pair was sent in (first format is with the "list" flag, while the second format is with the "rank" format).

  The first format is used to pass in the appropriate adjacency list. The second format passes in a value that is equal to the current node's rank divided by its outgoing degree:

$$[\text{NEIGHBOR\_RANK}] = \frac{[\text{CURRENT\_RANK}]}{[\text{ADJACENCY\_LIST}].\text{size}()}$$

The reason why we do this is twofold. First, it will avoid matrix multiplication, which is a very time-consuming process. Secondly, the PageRank value for each node in the graph is equal to:

$$r_j = \sum_{(j,i) \in E} \frac{r_i}{d_i}$$

By passing this value into the reduce function, we assume the sort would allow all of the NodeId's to be grouped together so that PageRanks can be calculated for each iteration. Furthermore, since the adjacency matrix is passed in as well, the idea is to have the adjacency matrix sorted at the beginning of each node block.

- PAGERANK_REDUCE: The input of this PageRank reduce function will be format denoted in the PAGERANK_MAP section. We also expect that the Amazon machine will sort based on NodeId, so the implementation that we have is to reset aggregate rankings and adjacency lists whenever a new NodeId is encountered. Since there are two different input styles, I will state what the reduce function will do for each type of input.
  - **Rank**:
    We expected the nodes all to be grouped up based on NodeId. So, with the knowledge of sort sorting "link" in front of "rank", we assumed that whenever we saw a "rank" in the key, then all subsequent keys for that node would also include "rank". Since this key/value pair would include a rank increment, we aggregated all of these rankings, and set that aggregated ranking to be equal to the new ranking of the node of that block.
  - **Link**:
    Since "link" is less than "rank" when you compare them, we assumed that for a particular NodeId group, the adjacency list will come first. Thus, if the reduce function receives a key/value pair with the key containing "link", it would simply store the adjacency matrix.

    There is one edge case: the adjacency matrix is empty. This means there are no outgoing edges, so instead of aggregating the rank starting at 0, we initialize the rank to be the current rank of that node (also passed in via key), and aggregate ranks from the "rank" key/value pairs.

  After retrieving the adjacency matrix and setting the current rank to be the aggregated rankings for a particular NodeId, the PageRank reduce function will output a key/value pair in the following format:

  NodeId:[NODEID] [ITERATION],[CURRENT_RANK],[PREVIOUS_RANK],[ADJACENCY_LIST]

  This output will then be passed into the PROCESS_MAP function.

- PROCESS_MAP:
  The process map function will take in key/value inputs with the above format, and from there, keep track of the top 20 rankings. This function stores a global variable known as "top", which stores the

top 20 rankings. For each key/value pair, the input node is compared with the rank 20 node, and if the input node has a higher ranking (or the top 20 list isn't fully populated), it is added to the list (and removing the 20th position if the list was previously populated). The list is then resorted in decreasing order. In the case that the node has been added to this top 20 list, the function will output another key/value pair for the reduce to handle:

A:[NODEID] [CURRENT_RANK]

Note: The reason why "A:" is added is for the sorting to keep these into account first. This will pose problems in the future, which we will list out in future sections.

If the input node has not been added, then we simply output the input line (so it would be the identity function).

- PROCESS_REDUCE:
  The process reduce function is pretty similar to the process map function. Like the process map function, there is a global list of the top 20 scores, except we add ranks into that list whenever we receive a key starting with "A:". The inputs for this function have keys that start with either: "A:" or "NodeId:". Since we originally assumed sorted/grouped keys for this, (which is wrong), this is what we had planned the program to do:
  - **A:**
    For each iteration, the sorting would cause this reduce function to process these first. Whenever a key starts with "A:", we would either append it to the list (if it wasn't originally populated), or replace the 20th rank with the input rank, and resort the list.
  - **NodeId:**
    If this were the case, the reduce function would either serve as the identity function (if the iteration count is not maximum), or we would print out its final rank (if the iteration count is maximum).

  Therefore, at the very end of the simulation, the output for reduce should be:

  FinalRank:[CURRENT_RANK] [NODEID]

Problems

While the idea behind our initial framework was largly correct, we erred on one key point: We relied too much on sorting for things to actually work. That is, our implementation is not order invariant. This is seen both in Process_Map and especially PageRank_Map. Furthermore, though our implementation worked for Amazon testing and local testing, it wouldn't work for multi machine testing, simply because of the way we organized the key/value pairs. By having different key values for the same NodeId, we cannot confirm that all NodeId's would really be grouped up during the PageRank reduce step, which is the crucial step for PageRank calculations.

For example, the keys in the key/value pairs are in the format:

[NODEID],[FLAG],[CURRENT_RANK],[ITERATION].

We addressed above that the flag value changes whether or not the key/value pair is for an adjacency list or for a rank. Because of different flags for the same NodeId, the adjacency list might be stored on another machine while all the ranks are stored elsewhere. This will lead to incorrect calculations, since wrong adjacency lists might be used. So, to move from a single machine multiple machines, we needed to fix the key-value pairs to be in the correct format. The progress for that is listed in the section below.

3 Refactored Framework

Refactoring

Unlike the previous framework, our refactored code kept iterations in a separate key value pair. So, the inputs to the PageRank Map functions were either:

NodeId:[NODEID] [CURRENT_RANK], [PREVIOUS_RANK], [ADJACENCY_LIST]

or

Iters [ITERATIONS]

Like the previous section, we will detail the refactored bits in each of the four functions. we will talk about what the original problems were, and how the refactored code fixed it.

- PAGERANK_MAP:
  As mentioned above, the problem here was that we made a poor mapping choice. Originally, our key values were [NODEID],[FLAG],[CURRENT_RANK],[ITERATION], but we wanted all of our key/value pairs to be blocked based on NODEID only. Therefore, the key value pairs were changed to:

  [NODEID] [FLAG],[CURRENT_RANK], [PREVIOUS_RANK], [ADJACENCY_LIST]

  or

  [NEIGHBORID] [FLAG],[NEIGHBOR_RANK]

  The first format of this key/value pair is if the [FLAG] is equal to "Adj:". This is pretty simple; we no longer include [ITERATIONS] and we just pass the [CURRENT_RANK] and [FLAG] to the values part. The second format is similar; and this occurs when [FLAG] is equal to "Rnk:". However, so that this function becomes order invariant, we use a dictionary to store each [NEIGHBOR_RANK] with the key being the [NEIGHBORID].

  Since [ITERATIONS] was no longer part of the mapping, we dealt with [ITERATIONS] as an entirely separate key/value pair:

  'Iters' [ITERATIONS]

  We left it separate from the node calculations, and only decided to check iterations at the very end. So, if we received an iterations key/value pair, we just passed it along.

- PAGERANK_REDUCE:
  The problem with our old framework was that we relied too much on sorting, and we were naive of Hadoop's intricacies for grouping. This issue was essentially fixed once we moved the [FLAG] and the [CURRENT_RANK] into values and left [ITERATIONS] to be a separate dataflow. Furthermore, we also used dictionaries to store adjacency lists, with they key being the current node. This was later optimized to string buffers, which will be explained below.

  We did notice, however, that some datasets that we mined did not include NodeId entries for nodes that did not have any outgoing edges. The given datasets that were given did include entries for no outgoing edges, but according to Piazza, there may also be cases where no entries exist for such nodes. In this case, we would simply just create a new key/value pair with that NodeId, with a self-loop, thereby adding an entry to the initial set of data.

  In addition, for nodes with no outgoing edges, and thus no adjacency list, we added a new an adjacency list to itself so that it would contribute completely to itself, by definition of pagerank.

  Just like PageRank Map, we do not deal with iterations here. We just pass them along.

  The output for this function is simply:

  NodeId:[NODEID] [CURRENT_RANK],[PREVIOUS_RANK], [ADJACENCY_LIST]

  or

  'Iters' [ITERATIONS]

- PROCESS_MAP:
  Initially, our process map and our process reduce were relatively similar.Furthermore, we only wanted to process our data once, so we moved all of the processing to process reduce (we had a huge bug where we assumed only one process map, which wasn't always true on Hadoop). So, this function is simply the identity function: it passes its input as its output.

- PROCESS_REDUCE:
  This is the function that keeps track of the top 20 final rankings. This is also where the iteration key/value pairs are created or updated, incrementing the iteration number by 1 before passing it out. We used a priority heap holding the top 20 values; this allows us to quickly check the minimum top-20 value (just by comparing with the top of the heap), and the data structure resorts itself in $O(logn)$ time.

  Once the maximum number of iterations is hit, we simply print out Final Ranks.

After this implementation, the refactored code worked for both single instances and multiple machines. Our first 50 iteration run gave us a score of approximately 4:55:XX, with 0 penalty.

Optimizations

During the development of our refactored code, to avoid order reliance, we would use dictionaries to store most of our data. For example, we would store adjacency lists in a dictionary, with the key being the NodeId for that dictionary. However, once we began to understand the intricacies of Hadoop more, we transitioned from using dictionaries to using string buffers, as dictionaries take up a lot of space (for such a large dataset). This optimized space and time as well, especially for such a large dataset.

We used dictionaries to preliminarily collect PageRank contributions in pagerank_map. This way, we would print far fewer lines in pagerank_map, resulting in significant speedup.

Other low level optimizations included hardcoding alpha values and unraveling some of the for loops from the initial framework, so that there would be a slight speedup in running the code.

Perhaps the largest optimization that we did was to account for the tradeoff between time and error. We noticed that each iteration took about 6 minutes to complete on the Amazon machines. To determine an ideal stopping point, a script was written to test various iterations and the amount of error that they would achieve on our Amazon dataset of 250,000 nodes. We found that 15 iterations was the ideal value. This significantly sped up the PageRank time, since the graph is so large, and the error produced by this optimization was very small (4 minutes). This is probably due to the fact that for a degree heavy-tailed distribution, most of the nodes we will be calculating PageRank for are sparse, and would not contribute much to the top 20 rankings. We leave 15 iterations, however, to account for some convergence of the top 20 rankings (our last upload used 10 iterations, and we found that the error was too high). Even though there will be error present, the tradeoff between error and time significantly favors time minimization as the number iterations increases, since error rapidly decreases as more iterations are done.

The result for this optimization was what allowed us to cross Milestone 1 and Milestone 2. For 15 iterations, our error was:

$$1:30:11 + 0:04:00 = 1:34:11$$

4 Testing

As mentioned above, one of the largest bugs that we found was that we did not account for Hadoop's intricacies in MapReduce. For our initial framework, we built a script (called "script.py") that would simulate the program on a single instance machine. The script would basically run

```
python pagerank_map.py | sort | python pagerank_reduce.py | sort |
python process_map.py | sort | python process_reduce.py
```

We were really confused initially since this script would produce the right result, and the Amazon testing also produced the right result. However, once we found out about the nature of Hadoop's multi-machine cluster, we wrote another script that was to simulate a multimachine PageRank simulation (called "multiMap.py"). So, essentially, we wrote a MapReduce script that would simulate Hadoop. For the purposes of PageRank, we simulated 10 machines.

What this script did was to first create 10 text files for each of the pagerank functions (to simulate the 10 machines), 1 or 2 text files for process map, and 1 text file for process reduce (since map and reduce were said to be single instance). The script would first intake the maximum number of iterations, and also the data file that we would like to operate on. It would then parse the data file into the 10 separate text files, and in each file, each NodeId would represent a block. From there, it would conduct PageRank Map through each of the 10 files, populate the PageRank Reduce text files, conduct PageRank Reduce, populating the process map files, and etc.

By running this script on our initial framework, we immediately saw our problem with running on multiple machines; this script was a key part for our debugging.

We also took advantage of data mining. Our team looked through several datasets, and we tested our implementations not only on "EmailEnron" and "GNPn100p05", but also on datasets taken from Stanford's SNAP repository, such as "Wiki", "Facebook" and "Amazon". These datasets were much larger, and contained edge cases that assisted with debugging. One such example is the omission of any NodeId entry for nodes with no outgoing edges (this was found in "Wiki").

By running these scripts on all 5 of these datasets, and through constant local and Amazon testing, we believed that our implementations were correct. Like mentioned above, we used both single-instance "script.py" and multi machine "multiMap.py", and confirmed that both gave equal values for our local testing. For optimizations, such as optimizing the number of iterations, we usually conducted them locally and compared the rankings to the final rankings in assignment.

5 Contributions

There are four members in this group: Kevin Chang, Timothy Chou, Albert Ge, and John Li. Here are each of their contributions:

- KEVIN CHANG: Primary optimizer. Researched the framework of Hadoop to help debug and refactor our initial framework to working code. Aided in the refactoring of our initial framework to working code.
- TIMOTHY CHOU: Refactored the initial framework to working code. Primary simulation and script writer for local testing. Implemented a local version of Hadoop through "multiMap.py".
- ALBERT GE: Primary developer of the initial framework for the Map Reduction. Researched and explored optimizations using Monte Carlo Random Walk simulations. Aided in writing the report.
- JOHN LI: Developer and debugger in the initial framework. Helped refactor the code to account for multiple machines, and for undirected graphs. Primary uploader and Amazon tester.

6 Other Algorithms/Attempts

1. One algorithm that was attempted was using a Monte Carlo random walk simulation to determine pagerank, described in a paper written by Sarma et al [2]. The basic idea is, at each node, start with

a fixed number of random walks, K. At each iteration, the random walks have a probability of α of taking a directed step to one neighbor, and probability $1 - \alpha$ of terminating. Each node would then keep track of the number of walks walking into it. The final pagerank calculation would be output as the number of walks visiting a node, divided by factor proportional to the size of the graph.

Initial Thoughts This algorithm appeared as a viable optimization over calculating pagerank through the transition matrix due to the initial fixed number of walks. The expected stopping time of a walk is $\frac{1}{\alpha}$, so the number of walks present monotonically decreases over iterations, and therefore the computation will gradually speed up (and in fact, this is what we saw in our implementation). Additionally, it could serve well in graphs with a heavy-tailed distribution; since nodes with a higher in-degree node are likely to receive more random walks, it would suffice to start with only a small number of walks at each node, since we would expect that the Monte Carlo simulation would aggregate a majority of the walks at nodes that are on the right side of a heavy-tail degree distribution.

Implementation and Results This algorithm didn't appear so great, at least for small graphs (e.g., the sample sets GNP and EmailEnron). Results only approximated the top 20 rankings for large initial random walks K, If K was too small, then there would be several nodes all with the same pagerank value - this appears especially problematic with the GNP dataset, since for the true pagerank values of the top 20, 19 of them contain pagerank values that are within 1 of each other. Thus, since K was so large, it took longer to run than our machine-level optimized transition-matrix computation, though we did notice that iterations sped up over time. Perhaps the results would be different if we included the same optimizations in the Monte Carlo simulation, but since our naive implementation had not been working until the near end of the week, we spent the remaining time debugging and testing that implementation, and thus this project yielded inconclusive results.

Other Considerations The paper also considered an improved version of the Monte Carlo method, but it would only hold for undirected graphs. This was not attempted, since early on in the week it was discovered that uploading a program which assumed graphs were undirected yielded results with high penalty.

2. The second optimization that was attempted was also using a design pattern written in a paper by Lin and Schatz [1]. In the paper, the authors describe a concept known as "Schimmy" to take advantage of the parallel mappers and reducers. This algorithm would partition the existing graph into the same number of partitions as the number of reducers; thus, every reducer would only operate on the same nodes. This would prevent the need to emit the entire graph structure within each iteration, as the reducers would know which nodes to update.

Thoughts This would require that the reducers somehow have access to the graph structure without obtaining it from the input data. Thus, it would be impossible to completely implement Schimmy, but one possible idea that was thought of was to emit a partition of nodes as a single line. This allows the number of read/writes between mapper and reducer to just be a fixed constant, which dramatically reduces traffic in the intermediate data between mapper and reducer. However, this would also require that the graph partition be sent to the reducer before any of the pairs containing updated ranks be sent. This would only occur if the values are also sorted in the intermediate step (since the keys could only be used to identify each reducer), so it was unclear if this was a viable

optimization.

References

[1] Jimmy Lin and Michael Schatz. "Design patterns for efficient graph algorithms in MapReduce". In: *Proceedings of the Eighth Workshop on Mining and Learning with Graphs*. ACM. 2010, pp. 78–85.

[2] Atish Das Sarma et al. "Fast distributed pagerank computation". In: *Distributed Computing and Networking*. Springer Berlin Heidelberg, 2013, pp. 11–26.