Rankmaniac Report

Team RankBeaver

Xinjie Lei, Jian Zhu, Tianlei Sun, Hongnian Yu
California Institute of Technology
Department of Electrical Engineering

1 Team Member & Work Split

- **Group members**
  Xinjie Lei
  Tianlei Sun
  Hongnian Yu
  Jian Zhu

- **Team name**
  RankBeaver

- **Division of labour**
  Xinjie Lei: Implemented the PageRank reduce algorithm, helped in debugging PageRank map algorithm, explored ways of optimizing code structure, and brainstormed for speed optimization.
  Tianlei Sun: Coordinated and debugged different modules, optimized the Neighbor List Algorithm, researched and implemented Top K Page Rank method.
  Hongnian Yu: Implemented the page rank mapper algorithm, helped debug page rank reducer program, searched paper online and brainstormed for speed optimization.
  Jian Zhu: Implemented the processor algorithm, helped debug page rank reducer program, and collaborated in optimizing the Top K Page Rank method.

2 Introduction

- Page Rank is an algorithm for measuring the importance of pages in a web graph. It is developed by Larry Page and Sergey Brin in 1996, and later adopted by Google for its initial prototype search engine. The algorithm takes in representation of a web graph (adjacency List, adjacency matrix, etc..) and output the probability distribution for different web pages. The details of page rank algorithm is given as follows:

$$G = \alpha * P + \frac{(1-\alpha)}{n} * \mathbb{1},$$

$$r(t) = r(t-1) * G$$

n is the total number of vertex in the graph, and alpha is the damping factor, P is the transition matrix, r(t) is the page rank vector at iteration t. We initialize r(0) to be 1/n, and multiply it by G at each iteration.
The map reduce framework comes into place if we realize that the page rank of node i at iteration t, denoted as r(t)[i], is only determined by the transition matrix and page rank vector at the previous iteration. Therefore, we can calculate the page rank of different node at iteration t in parallel. Moreover, considering r(t)[i]=r(t-1)*column i of G, this computation process is equivalent to summing up all the contributions to node i from all the node which are connected to i. In this way, we no longer need to deal with matrix form multiplication. The implementation details can be found in the algorithm section below.

3 Original Algorithm

- **Mapper Algorithm**
  - **Pseudo code:**

Algorithm 1 Mapper algorithm

```
1: procedure MAPPER($std\_input$)                    ▷ mapper actual input is inside std_input
2:     $(nodeID, iteration, currRank, neighborsList) \leftarrow parse\ std\_input$
3:     for $neighbor$ in neighborsList do
4:         $individualScore \leftarrow \alpha * currRank/length\ of\ neighborsList$
5:         $std\_out.write$(neighbor, individualScore)
6:     end for
7:     $std\_out.write$(nodeID,iteration,neighborsList)
8: end procedure
```

  - **Explanation:**
    The mapper function first read and parse std input. Then we calculate the individual score based on the formula $individual\ score = \alpha * currRank/out\_degree$. Individual score means the contribution of node i to its neighbor j for j's page rank in the current iteration. We emit neighborID and contribution. This contribution is later collected by collectors and group by key(neighbor's id) to determine the neighbor node's page rank in the current iteration. In addition to individual score. We also need to emit nodeID, iteration number, and its neighborsList so that the process reducer can keep track of the current iteration number and not lose the graph information(nodeID and neighborsList). To improve efficiency, this information only need to be emit once.

- **Reducer Algorithm**
  - **Pseudo Code:**

Algorithm 2 Reducer algorithm

```
1: procedure REDUCE
2:     $key$ ←None
3:     $rankContributions$ ←{}
4:     for $line$ in std_in do
5:         $nodeID, oneContribution, iteration, neighborsList$ ← parse($line$)
6:         if key is None then
7:             $key$ ←nodeID
8:         else if $key$ ≠nodeID then
9:             $rank$ ← sum($rankContributions$) + 1 - $\alpha$
10:            std_out.write($nodeID, rank, iteration, neighborsList$)
11:            $rankContributions$ ←{}
12:            $key$ ←nodeID
13:        end if
14:        append $oneContribution$ to $rankContributions$
15:    end for
16: end procedure
```

– **Explanation:**
Based on the PageRank algorithm, we derived the following formula for PageRank of an individual node,

$$r(t)_{i,j} = \alpha * r(t-1)_{i,*} * P_{*,j} + n * \frac{1-\alpha}{n} = \alpha * r(t-1)_{i,*} * P_{*,j} + 1 - \alpha$$

The mapper already emits $r(t-1)_{i,*} * P_{*,j}$ element-by-element in a sequence. The collector groups them and sends the sequence to the reducer. Thus, the reducer only needs to sum up the PageRank contributions from all neighbors of key(node id), multiply the sum with the damping factor α and add the result with $1-\alpha$ which guarantees an aperiodic and strongly-connected graph.

• **Processor Algorithm**

– **Pseudo Code:**

Algorithm 3 Processor algorithm

```
procedure PROCESS
    $topRank \leftarrow \{\}$
    for $line$ in std_in do
        $nodeID, rank, iteration, neighborsList \leftarrow$ parse($line$)
        if $iteration <$ MAX_ITR then                         ▷ More iterations to go
            std_out.write($nodeID, rank, iteration, neighborsList$)
        else
            while length($topRank$) $< 20$ do       ▷ use heap sort to find the top 20 ranks
                add $(nodeID, rank)$ to $topRank$
            end while
            $(id, minVal) \leftarrow \min(topRank)$
            if $rank > minVal$ then
                delete $(id, minVal)$ from $topRank$
                add $(nodeID, rank)$ to $topRank$
            end if
        end if
    end for
    sort the top 20 ranks
end procedure
```

– **Explanation:** The processing part is executed after mapping and reducing. It takes the output from the reducer algorithm and decide whether to proceed to another iteration or output the final results. If the maximal iteration which is set manually has not been reached yet, it will parse the input to the format that can be read by the mapper. Otherwise it will use heap sort to find the top 20 ranks and then sort them in descending order as the output.

4 Optimization

- **Code Structure Optimization:**
  We considered following code structure optimizations:
  – Avoid unnecessary string conversion: For variables node id, iterations, and neighbor lists, we converted them to integers only when the variables participated in computations. Otherwise, we used string format of the variables for branching. This optimization improved the performance in a limited extent.
  – Avoid unnecessary list join: PageRank mapper receives neighbor node list for each node and passes it to the reducer. In the original code, we split the neighbor list string to a list and then joined the list to a new string as part of the standard output. This operation was inefficient because the large amount of nodes in test data set would introduce repetitive split and join in

each iteration. Thus, we kept the neighbor list string and created a new list variable holding the string splitting result to improve the overall long-run performance.

- Use regular expression for string parsing: We used several string splitting function calls in the initial line parsing method which created multiple unnecessary temporary lists. To improve the speed and memory utilization, we adopted regular expressions to parse the standard input considered the flexibility and conciseness of regular expressions. However, overall performance was worsen because of the inner greedy algorithm regular expression used to match a pre-defined pattern. Thus, we abandoned the regular expressions and stuck on string splitting instead.

• **Neighbor List Optimization:**
The main function of the Pagerank reducer part is to sum up the PageRank contributions from each nodes. However, it's also important to pass the neighbor list to the next iteration. In the beginning, we planed to attach the neighbor list at the back of each mapper emit. However, this is every inefficient because we only need one copy neighbor list for each node, while this method provide redundant copies of the neighbor list. Therefore, we brainstormed a new way with two different form of mapper output and reducer input:

1. NodeID -tab- value
2. NodeID -tab- iteration, neighbor list

The first form appears in each mention in neighbor list, but the second form appears once once. The reducer recognizes the two different formats, sums the values for each NodeID and output one result for each NodeID:

1. NodeID -tab- sum, iteration, neighbor list

This method dramatically increases the performance and helped us exceeds the milestone performance.

• **Top K page rank Optimization – An interesting failure attempt:**

- **Implementation:** After we pass the milestones, our next approach is to sacrifice accuracy for speed. We plan to reduce the time cost of each iteration. And our approach is to cut the unnecessary nodes affected by [1].

Algorithm 4 Top K page rank algorithm

```
procedure NODE CUTTING($graphList, K$)    ▷ graphlist is a list of all nodes
    $n \leftarrow length(graphList)$
    for $node$ in $graphList$ do
        if $node$'s rank is in top K then
            $topKlist$ appends $node$    ▷ Nodes in topK_list won't be cut
        end if
    end for
    $finalList \leftarrow topKlist$
    for $node$ in $topKlist$ do
        for $supportNode$ in $node$'s neighbor list do
            if $node$'s rank is in top K then
                $finalList$ appends $supportNode$    ▷ support Nodes of topK_list won't be cut
            end if
        end for
    end for
    return $finalList$    ▷ return the top K nodes and their support nodes
end procedure
```

– **Result:** This algorithm can reduce 50% of time cost in each iteration according to our local test. But unlucky, the high penalty of the AWS overhead(5min each iteration) means that the actual runtime of each iteration is only about 1min and thus there is no help in reducing time for each. We sadly find out that the only effective way is to reduce the number of iterations and predict the result.

5 Conclusion

- **Discoveries**
  In this project we implemented MapReduce to compute the ranks of nodes in a graph based on the transition matrix. We realized that it could be time-consuming to iterate over the transition matrix until it converges, so stopping criterion and optimization on the computation process would be meaningful. We found that generally it will converge after a dozen times, so reducing the number of iterations or the running time in each iteration would make the process much more efficient. We tried the method of optimizing the page rank algorithm and found it effective in lower the time cost of each iteration. However, due to the AWS overhead, the progress was not so dictinct in practice.

- **Challenges**
  The main challenge was that our optimization was limited by the overhead in each iteration. Our attempt in optimization was effective in the local test, but we did not realize the overhead issue. Stuggling with decreasing time cost in each iteration instead of reducing iterations proved to be not so effective in AWS although we tried multiple methods in optimizing our algorithm.

- **Future Improvement**
  We would consider reducing the number of iterations to optimize our algorithm. This is more pratical regarding the overhead in each iteration. By setting some stopping criterion and predicting principle we can terminate the computation process earlier and predict the final ranks with enough accuracy. This would be much more beneficial if we take the actual running cost into consideration.

6 References

[1]. Fujiwara, Y., Nakatsuji, M., Shiokawa, H., Mishima, T., Onizuka, M. (2013, July). Fast and Exact Top-k Algorithm for PageRank. In AAAI.