The GitHub map

Lucas Bergeron and Ali Karooni
Supervised by Tomasz Wiktorski

April 2019

Abstract

In the context of DAT500 courses, the final project is an important step for students. It permit to put in practice the theoretical notions see in class, applied to a concrete project. This project is realized by group of two.
The students have to find a dataset to process MapReduce algorithm and extract relevant results from it. Spark and MrJob libraries are the main tools used during the project.

Keywords : Python, PageRank, MrJob, Spark, GitHub

1 Introduction

Traditional maps are used for self-orientation or data analysing based on the world geography. It could be a territory, a building or even a network. But when we have to create a visualize representation of something virtual, things becomes more complex.

How can we plot a map or data without any axis or geographical location ?

The Internet Map [1] The Internet map is a scheme displaying objects relative position; but unlike real maps or virtual maps, the objects shown on it are not aligned on a surface. Mathematically speaking, The Internet map is a bi-dimensional presentation of links between websites on the Internet. Every site is a circle on the map, and its size is determined by website traffic, the larger the amount of traffic, the bigger the circle. Users switching between websites forms links, and the stronger the link, the closer the websites tend to arrange themselves to each other.

The GitHub Map [2] Inspired from the Internet Map, the GitHub Map have for objective to find the most relevant users and repositories on GitHub and create a map of the community. For this, we gonna process PageRank algorithm to give as score, or rank to each user and repository.

Contribution Summary This report will talk about the following items :

- The PageRank algorithm either iterative and using MapReduce structures.
- Implementation of PageRank algorithm using MrJob and Spark.
- The input file used for the evaluation and the corresponding results.

2 Page rank algorithm

PageRank is an algorithm used by Google Search to rank websites in their search engine results. The algorithm works by counting the number and quality of links to a page to determine a rough estimate of how important the website is. The underlying assumption is that more important websites are likely to receive more links from other websites.

We are going to use the same idea in this project. However, GitHub repositories and users are nodes linked by the number of commits which the user have done on the repository.

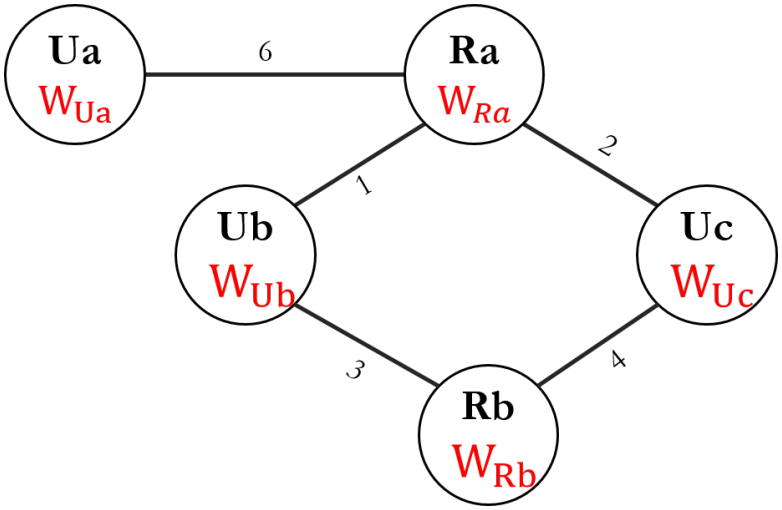

Figure 1: Simple graph example.

Lets assume we have the following graph, Figure 1. Three users *(Ua, Ub and Uc)* are contributing to 2 repositories *(Ra and Rb)*. We can calculate the percentage of contribution for each user on a repository and opposite.

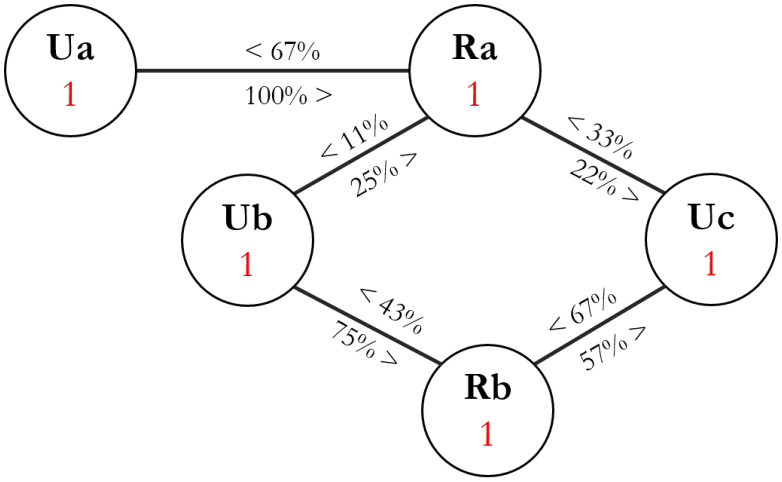

Figure 2: Initial PageRank graph.

We start by setting to one the score of each node. At each iteration of the PageRank algorithm, the score of a node will be transferred to the connected nodes regarding the percentage of contributions.

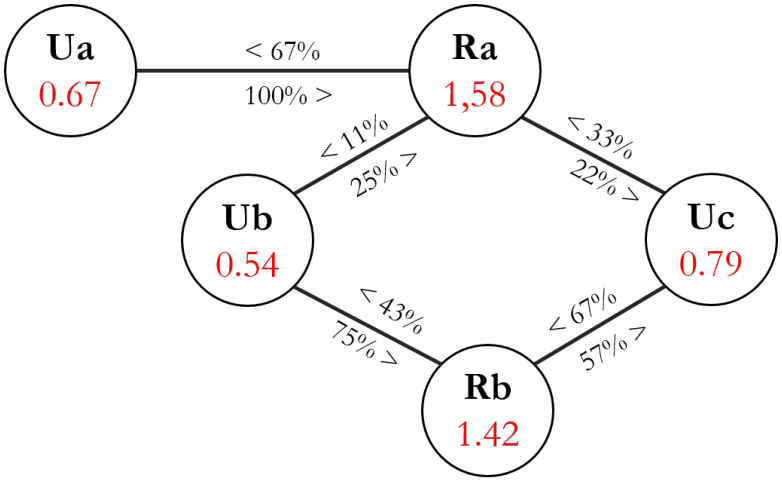

Figure 3: Resulting PageRank graph after the first iteration.

After each iteration, the new scores are compute, Figure 3. The process is repeated again and again until convergence. We say that the graph have converged when the difference between the current score and the previous score is less than a threshold.

2.1 Iterative algorithm

We can model the PageRank algorithm with the following sequence where W is the score matrix, M the links matrix and d the damping factor.

$$W_{n+1} = M.W_n.d + (1-d) \quad W = \begin{pmatrix} W_{R_a} \\ W_{R_b} \\ W_{U_a} \\ W_{U_b} \\ W_{U_c} \end{pmatrix} \quad d = 0.85$$

$$M = \begin{pmatrix} M_{R_aR_a} & M_{R_aR_b} & \cdots & M_{R_aU_c} \\ M_{R_bR_a} & M_{R_bR_b} & \cdots & M_{R_bU_c} \\ \vdots & \vdots & \ddots & \vdots \\ M_{U_cR_a} & M_{U_cR_b} & \cdots & M_{U_cU_c} \end{pmatrix} = \begin{pmatrix} 0 & 0 & \cdots & 0.33 \\ 0 & 0 & \cdots & 0.67 \\ \vdots & \vdots & \ddots & \vdots \\ 0.22 & 0.57 & \cdots & 0 \end{pmatrix}$$

Figure 4: PageRank algorithm using matrix.

If N is the number of nodes in the graph, the W matrix is a column matrix of size N and M is a scare matrix of size N. Implementing the PageRank algorithm in that way, is really simple by takes a lot of memory. As you can see on the Figure 4, the M matrix is partially full. A lot of zero values are stored while they are not relevant for use. Moreover, an iterative algorithm cannot be parallelized.

2.2 MapReduce algorithm

The MapReduce structure permit to save memory during the process and the tasks can be parallelized. Here is how we will implement the PageRank algorithm using mapper and reducer.

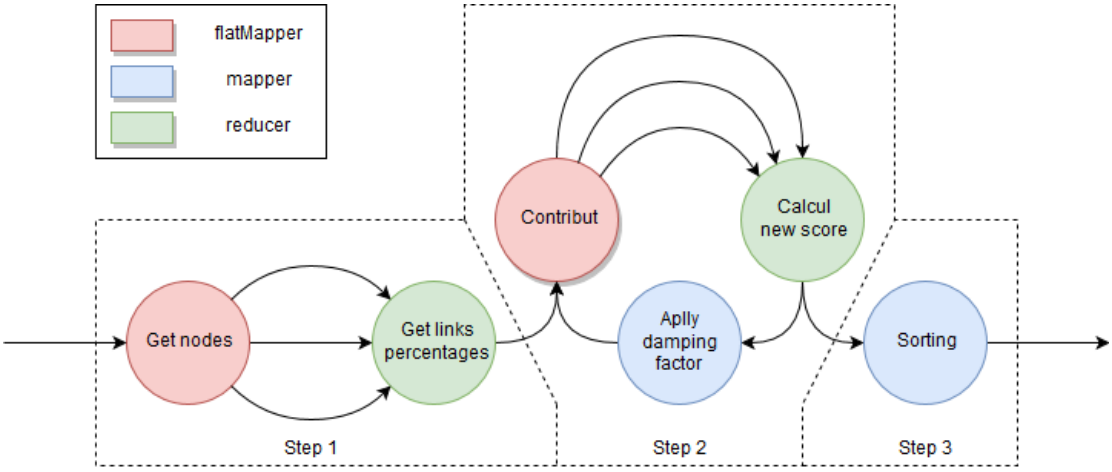

Figure 5: PageRank flow diagram.

Step 1 Consist of extracting the links from the input file. *We will discuss about the structure of this file in the next section.* One mapper and one reducer are needed to list for every nodes the percentage of contribution on the connected ones.

Step 2 This is the PageRank algorithm step : For each connected nodes, a part of the score is gave to the connected nodes based on the percentage of contribution. The reducer sum all the received values and apply the damping factor before processing again.

Step 3 After a certain amount of iterations, the ranks are sorted before outputting.

3 Implementation

We have implemented the PageRank algorithm either using MrJob and Spark. For both implementations, we have to remind the links input file structure.

```
repoA   userA   12
repoB   userA   4
repoA   userB   16
repoA   userB   20
...
```

Figure 6: Links input file.

As you can see on Figure 6, each line of the input file represent the number of commits for a user on a repository. The first step will always consist of listing for every nodes a dictionary representing the percentage of contributions.

```
"repoA" : {"userA": 0.25, "userB": 0.333, "userC": 0.417}
"userA" : {"repoA": 0.75, "repoB": 0.25}
...
```

Figure 7: Task 1 output.

The Figure 7 represent the dictionary created regarding the Figure 6 input file.

3.1 MrJob

This section explain how to implement the PageRank algorithm step-by-step using MrJob.

```
def get_commits_mapper(self, key, value):
    values = value.strip().split('\t')
        if len(values) == 3:
                repo = values[0]
                email = values[1]
                commits = values[2]
                yield repo, (email, int(commits))
                yield email, (repo, int(commits))

def get_commits_reducer(self, node_name, values):
                node = {'key' : node_name}
                links = {}
                tab_values = [value for value in values]
                total_commits = sum([value[1] for value in tab_values])
                for value in tab_values:
                        links[value[0]] = value[1]/total_commits
                node['links'] = links
                yield node, 1
```

Figure 8: MrJob - Step 1 - Create nodes with links dictionary.

The first step consist of extracting nodes from the links input file and listing every connected nodes. For that, one mapper and one reducer are needed.

Mapper In the mapper, two keys are yield for one line. The first one is the user and the other one is the repository. Both of them are nodes linked with a number of commits. This number is included in the corresponding value as well as the name of the connected node. *Example : If UserA is connected to RepoB by 2 commits, then two keys are yields. UserA : (RepoB, 2) and RepoB : (UserA, 2).*

Reducer Once all links have been processed, we reduce them to get all the connected nodes for one node. We can now create for each node a dictionary composed of a name and a list of connected node with the percentage of contribution for each. *Example : If UserA is connected to RepoB by 2 commits and RepoC by 6 commits, the total number of commits is 8. The percentage of contribution on repoB is 2/8=25% and 6/8=75% for repoC.*

The output value for each node is the default rank value. In that case, we have choose 1 so that the score of the graph is always equal to the number of nodes N.

```
def pagerank_mapper(self, node, weight):
        yield {'key' : node['key']}, node['links']
        neighbours = node['links']
        for neighbour in neighbours.keys():
                yield {'key' : neighbour}, weight*neighbours[neighbour]

def pagerank_reducer(self, node, values):
        weight = 0
        for value in values:
                if(type(value) == dict):
                        node['links'] = value
                else:
                        weight+=value
        yield node, weight*self.options.damping_factor+(1-self.options.damping_factor)
```

Figure 9: MrJob - Step 2 - PageRank single step.

Here is the most important step in the algorithm, the PagRank step. One step consist of a mapper and a reducer. The mapper and the reducer on Figure 9, are called multiple times.

Mapper The mapper distribute for each connected node a part of the node rank. But it is interesting to note that the links are also yield. We cannot yield the node name and the connected nodes list as a dictionary like in the previous step because we don't have access to the neighbours connected nodes list. The idea is that the nodes give to themselves the connected nodes list. We just have to distinguish the links values from the rank values in the reducer.

Reducer In the values list, we know that one of them contain the links as a dictionary. So, if the value is a dictionary type, it's stored in the key. All the other values are summed to get the new score.

```
def output_mapper(self, node, weight):
        yield None, (weight, node)

def output_reducer(self, _, nodes):
        for weight, node in sorted(nodes, reverse=True):
                yield (weight, node['key'])
```

Figure 10: MrJob - Step 3 - Sorting nodes by rank.

In the last step, Figure 10, we just want to sort the node by descending scores. We need a list containing all the scores on one machine. In the mapper, the weight and the node name are yield for a constant key. So that in the reducer, we are sure that all the values are compute on a single machine and that the values list contain every scores. This step is critical for memory since there is no parallelization.

3.2 Spark

We will see in this section how to implement a PageRank algorithm using Spark. The idea is similar as the previous implementation with MrJob. However, Spark offer a huge advantage in comparison to MrJob. The result of each task in Spark is accessible via a variable. That's because when the output of a task needs to be passed to another task, Spark send the data directly without using the persistent storage. In the case of PageRank algorithm, we will use this advantage to simplify the algorithm.

```
def getNodes(line):
        email = line.split('\t')[0]
        repo = line.split('\t')[1]
        commits = int(line.split('\t')[2])
        yield repo, (email, commits)
        yield email, (repo, commits)

def getLinks(node):
        name = node[0]
        neighbours = node[1]
        links = {}
        total_commits = 0
        for i in range(1, len(neighbours), 2):
                total_commits += neighbours[i]
        for i in range(0, len(neighbours), 2):
                links[neighbours[i]] = neighbours[i+1]/total_commits
        return name, links

nodes = links.filter(lambda line: len(line.split('\t')) == 3) \
        .flatMap(lambda line: getNodes(line)) \
        .reduceByKey(lambda a, b: a+b) \
        .map(lambda node: getLinks(node))
ranks = nodes.map(lambda node: (node[0], 1))
```

Figure 11: Spark - Step 1 - Create node with links dictionary and set default score.

During the first step, the idea is to find for each node every connected nodes, called neighbours. As shown in Figure 7 the output of the first step is a dictionary of connected nodes with percentage of contributions. The last line give to each node a default score of one.

```
def setWeight(node):
        for neighbour in node[1][0].keys():
                yield (neighbour, node[1][1]*node[1][0][neighbour])

for i in range(iterations):
        ranks = nodes.join(ranks) \
                .flatMap(lambda node: setWeight(node)) \
                .reduceByKey(lambda a, b: a + b) \
                .mapValues(lambda rank: rank*DAMPING_FACTOR + (1-DAMPING_FACTOR))
```

Figure 12: Spark - Step 2 - PageRank single step.

During the PageRank step, the results of the previous task are joined to create a tuple value composed of a dictionary and the current score. In the flatMap step, we just have to distribute the current score for each neighbour regarding his percentage of contributions. After that, we can reduce all the values for one node by summing them. The damping factor is next used in a mapValue to avoid non-stable systems.

```
DAMPING_FACTOR = 0.85
def getNodes(line):
        email = line.split('\t')[0]
        repo = line.split('\t')[1]
        commits = int(line.split('\t')[2])
        yield repo, (email, commits)
        yield email, (repo, commits)

def getLinks(node):
        name = node[0]
        neighbours = node[1]
        links = {}
        total_commits = 0
        for i in range(1, len(neighbours), 2):
                total_commits += neighbours[i]
        for i in range(0, len(neighbours), 2):
                links[neighbours[i]] = neighbours[i+1]/total_commits
        return name, links

def setWeight(node):
        for neighbour in node[1][0].keys():
                yield (neighbour, node[1][1]*node[1][0][neighbour])

def main(input, output, iterations):
        sc = pyspark.SparkContext(appName="pagerank")
        links = sc.textFile("file://" + input)
        nodes = links.filter(lambda line: len(line.split('\t')) == 3) \
                .flatMap(lambda line: getNodes(line)) \
                .reduceByKey(lambda a, b: a+b) \
                .map(lambda node: getLinks(node))
        ranks = nodes.map(lambda node: (node[0], 1))
        for i in range(iterations):
                ranks.saveAsTextFile(output + "/" + str(i))
                ranks = nodes.join(ranks) \
                        .flatMap(lambda node: setWeight(node)) \
                        .reduceByKey(lambda a, b: a + b) \
                        .mapValues(lambda rank: rank*DAMPING_FACTOR + (1-DAMPING_FACTOR))
        ranks.sortBy(lambda node: node[1]).saveAsTextFile(output + "/" + str(iterations))
        sc.stop()
```

Figure 13: Spark - PageRank full algorithm.

The resulting code is the following shown on Figure 13. We finally output the result after sorting it by rank.

4 Experimental Evaluation

4.1 Links Dataset

The database we are using [3] is about 3Tb queryable with BigQuery [4]. From this database, only the commits table is relevant for use. Here is an example of the table structure :

...	commiter	repo_name	...
...	{email:example@gmail.com, name:Example, .}	[repoA, repoB, .]	...
...	{email:hello@gmail.com, name:Hello World, .}	[repoB, repoN, .]	...

Figure 14: Commits table from GitHub database.

The size of the commits table is about hundreds of Gigabytes. We might not be able to download the committer and repository columns without running out of memory. But the data linking a user and a repository is not in the way we want to process it. Therefore, we have to find a smart query to reduce the amount of downloaded bytes.

```
"""WITH links AS
    (SELECT committer.email, repo_name
    FROM `bigquery-public-data.github_repos.commits`)
SELECT email, repo, COUNT(email) AS commits
FROM links
CROSS JOIN UNNEST(repo_name) AS repo
GROUP BY email, repo;"""
```

Figure 15: SQL query to lower the result size.

The query in Figure 15 list every user contributions repository with the number of commits linking them. The downloaded bytes fall from 120Gb to 3Gb. We can now easily create the MapReduce input file.

Three input files have been created :

- All links : 49,206,697 (3.32 Gb - 100%)
- Links >50 commits : 4,980,520 (387Mb - 10.1%)
- Links >100 commits : 3,397,220 (266Mb - 6.9%)

4.2 Results

We have process PageRank algorithm based on links with more than 100 commits. The results are quite surprising at first.

Rank	Repo	Score
1	MrAlex94/Waterfox	367.31
2	shenzhouzd/update	282.67
3	jrobhoward/SCADAbase	112.32
4	kunj1988/Magento2	86.40
5	xorware/android_frameworks_base	80.73
6	tarunprabhu/DragonEgg	76.53
7	meego-tablet-ux/meego-app-browser	44.63
8	wistoch/meego-app-browser	43.68
9	xen2/mcs	41.96
10	orumin/openbsd-efivars	40.46

Figure 16: Top 10 repositories.

Rank	User	Pseudo	Score
1	Barry Clark	barryclark	1473.90
2	Hakim El Hattab	hakimel	1002.71
3	clowwindy	clowwindy	758.55
4	Michael Rose	mmistakes	715.00
5	Gerrit Code Review	GerritCodeReview	580.55
6	Will Farrington	wfarr	528.61
7	Guillermo Rauch	rauchg	409.58
8	Fabien Potencier	fabpot	408.16
9	Durran Jordan	durran	359.46
10	Robby Russell	robbyrussell	355.38

Figure 17: Top 10 users.

We where expecting *torvalds/linux* repository in the top 10. Regarding the GitHub website, *torvalds/linux* have more contributors and commits than *MrAlex94/Waterfox*. But after verification, the links file contain less contributions for *torvalds/linux* than *MrAlex94/Waterfox* either for the full one and the samples. The result might not be representative of the current GitHub community because of the dataset, but the PageRank algorithm is working fine.

As you can see, the algorithm consider the first user three times more important than the first repository. Keep in mind that the results doesn't represent the most popular user or repository. In fact, the users that contribute to multiple well ranked repositories will receive more score than the user that contribute to the first repository. This is why *MrAlex94* and none of the top 10 repositories owners are in the top 10 users.

The top 10 users are those who contribute to the most repositories regarding the dataset. The bigger are the repositories scores, the bigger the user score is. The same logic can be applied to the repositories.

4.3 Performance Comparison

Library\Input	>50 commits	>100 commits	All links
MrJob	*Out of memory*	*Out of memory*	*Out of memory*
Spark	09:54	06:00	*Out of memory*

Figure 18: MrJob and Spark performance comparison for 20 iterations.

Unfortunately, we where not able to run PageRank algorithm using MrJob because the machine run out of memory. But the samples are working fine with the Spark algorithm.

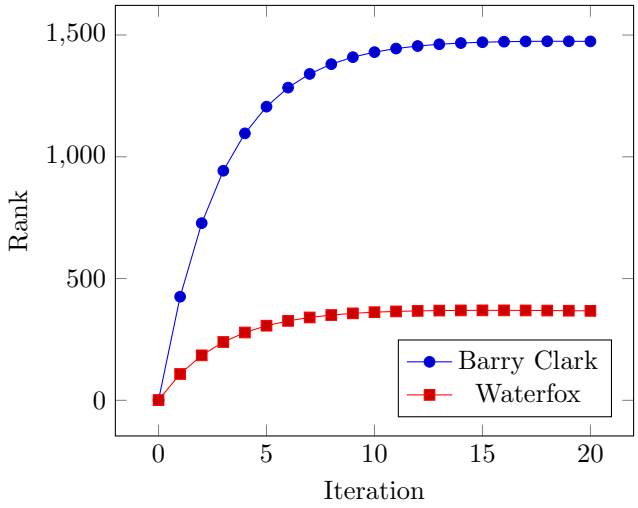

Figure 19: Barry Clark and Waterfox rank evolution.

The Figure 19 shows the evolution of the score for two nodes for 20 iterations.

5 Conclusion

The PageRank algorithm, is a powerful tool to score nodes in graphs. Combined with a Spring algorithm, it becomes possible to plot a graph data structure based on links connecting nodes. Because of a high number of nodes, it is not optimized to compute PageRank using a matrix. MapReduce structures are saving memory and make the algorithm work accurately.

Only the PageRank algorithm have been implemented in MapReduce. Our final goal isn't reached yet, improvements have to be done in optimization.

Even if the objectives of the project hasn't be reached, this project is a personal success for both of use. We have apply our knowledge in data intensive systems to extract relevant information from a dataset by using MapReduce algorithm. We have learn how PageRank algorithm works and even more during the entire year.

This is not the end of this project, only the beginning.

References

[1] R. Enikeev, "The internet map," 2011.

[2] LucBerge, "Github-map," 2019.

[3] "Github repos dataset - kaggle," 2017.

[4] "Bigquery api - google."