A personal recommender system based on an unweighted graph

Daniele Dellagiacoma and Davide Lissoni
Department of Information Engineering and Computer Science
University of Trento
via Sommarive 14, 38123, Trento, Italy
daniele.dellagiacoma@unitn.it, davide.lissoni@studenti.unitn.it

ABSTRACT
Nowadays, many organizations, companies and researchers need to deal with big datasets in the order of terabytes or even petabytes. A popular data processing engine for handling such big datasets in an efficient way is Hadoop MapReduce. These big datasets are often represented as graphs by many systems of current interest (i.e. Internet, social network). A key feature of these systems is provided by personalized recommender systems for information retrieval and content discovery in today's informationrich environment. Usually modern recommendation systems use complex technique to provide advices to each user. In this paper we explain our implementation of different way to provide recommendations to users of a network based on an unweighted graph. Using an Hadoop iterative MapReduce approach for the implementation.

Keywords
Big Data; MapReduce; Hadoop; Breadth-first search

1. INTRODUCTION
MapReduce is a programming model and an associated implementation for processing and generating very large data sets with parallel and distributed algorithm on a cluster. MapReduce has been ideated by Google's researcher Jeffrey Dean and Sanjay Ghemawat in 2004 for handling large data sets which are usually computed by hundreds or thousands machines to finish in the shortest possible period [1].

MapReduce works through two functions called Map() and Reduce(). The Map() function takes an input element and generated a set of intermediate key-value pairs and passes it to the Reduce() function. The Reduce() function takes the set of the intermediate key-value pairs, merges all the intermediate values for a particular key and produces a smaller set of merged output values.

The power of MapReduce is that it allows programmers without a deep experience in parallel and distributed systems to easily utilize the resource of a large distributed system, breaking a computation into small tasks that run in parallel on multiple machines, and scales easily to very large clusters of inexpensive commodity computers. Moreover, another key benefit of MapReduce is that it automatically handles failures, hiding the complexity of fault-tolerance from the programmer. In fact if a node crashes, MapReduce runs the task on a different machine [2].

A popular implementation of MapReduce is the open-source framework Apache Hadoop [3]. It was developed mainly by Yahoo! though is also used by other companies, such as Facebook, Amazon and Last.fm [4]. One of the characteristic of Hadoop is the use of HDFS (Hadoop distributed file system) which was designed to store a massive amount of data, in order to optimize storage and access operations to a small number of large file, rather than a lot of small file [5].

Nowadays, many systems of current interest to the scientific community can usefully be represented as graphs [6]. Some examples include the world-wide web, the Internet, social networks, citations of papers, food webs, biochemical networks and many more. Each of these networks consists of a set of nodes representing, for instance, computers or routers on the Internet or people in a social network, connected each other by edges, representing data connections between computers, friendships between people, and so forth. Often these graphs are made up of a huge amount of nodes and edges.

In the following sections is described our idea and the implementations of work that was done. We thought about a different way to provide recommendations to users of a networks. Our program aims to visit a unweighted graph starting from a set of nodes (also called keynodes) to find out which nodes can be reached by the keynodes within a adjustable maximum distance. It has been implemented using MapReduce in order to handle big datasets.

This paper is organized as follows. Section 2 describes related works which has been used as starting point for our implementation of the project. Section 3 presents the problem we are trying to solve whereas section 4 provides our solution and implementation. Section 5 describes all the experiments we performed using different sets of data. Finally, we conclude in section 6.

2. RELATED WORK
Our program is based on two well-known algorithms used for traversing or searching graph data structures. The first one is the Breadth-first search (BFS) algorithm. BFS is a graph search algorithm that aims to expand and examine all nodes of a graph by systematically searching through every solution. The search starts from a source node and the neighboring nodes are visited until there are no more possible nodes to visit [7]. One way of performing the BFS is by coloring the nodes and traversing according to the color of the nodes [8].

There are several types of shortest path-problems including single-pair, all-pair, single-source and single-destination shortest path problems. There are several algorithms that implement each of these types. The one which has been useful in our works is the Dijkstra's algorithm. It is an algorithm for finding the shortest paths between nodes in a graph. For a given source node in the graph, the algorithm finds the shortest path between that node and every other [9]. It can also be used for finding the shortest paths from a single node to a single destination node by stopping the

algorithm once the shortest path to the destination node has been determined.

There are a lot of different implementations of breadth-first search graph using Hadoop MapReduce [10] [11]. We used one in particular as starting point for our work. We started using the MapReduce BFS "Iterative MapReduce and Counters" developed by the users of the Hadoop Wikispaces [12]. This algorithm starts just from one node to visit the graph and stops only when the all the node in the graph has been visited. This implementation, as other found on the Web, differs from our idea in some key points. We need to start from multiple nodes and perform a BFS simultaneously from each starting node.

Although the performances of our implementation are quite the same of the original algorithm when the graph is not too big, the performance decreases when the number of keynodes is increased. In fact the original Hadoop Wikispaces algorithm, with a large number of keynodes, converges before our implementation. Despite that if the number of keynodes and the number of step are opportunely adjusted our algorithm works as fast as the original Hadoop Wikispaces BFS algorithm.

We preferred Hadoop rather than more recent framework such as Spark or Cassandra thanks to some Hadoop feature. Hadoop MapReduce is a more mature platform and its ecosystem is currently bigger thanks to many supporting projects, online material, tools and cloud services [13]. Another relevant feature of Hadoop MapReduce is that everything gets written to disk, including all the interim steps. In fact each output of an intermediate step can be used both to check the evolution of the algorithm and to provide some useful information. These information can be used to provide advices to users of the network even if the program is still running.

3. PROBLEM STATEMENT

Many system that are regularly used now can be described as graphs. Some of them are social networks, world-wide web; Internet and many other. Many of those system use a method to provide suggestions to the users of the network. Some example are YouTube [14] that suggests you videos to watch, Facebook recommends people that you could know, Amazon [15] suggests similar product you bought and so on and so forth.

Often those recommender systems use complex technique to provide advices to each user. The advices can be based on measurements of similarity between items or users. Existing recommender systems can be categorized into two different groups: content-based and collaborative filtering [16]. In content-based approach the system recommends items to a specific user, similar to previous item rated highly by him/her. In this case every item is represented using a vector of features where value of a feature can be established by TF-IDF. In collaborative filtering, however, the system builds a database of preferences for items by users. A new user is matched against the database to discover neighbors, which are other users who have historically had similar taste to him/her. Items that the neighbors like are then recommended to the user, as he/she will probably also like them [17]. Some methods used to find similar users or items are memory-based techniques, model-based techniques and hybrid techniques.

We are trying to provide an alternative way to marking suggestions in different kind of systems (i.e. social network, e-commerce, etc) where the structure of the system is described only as a graph. Our model needs a graph and a set of keynodes. The graph may be directed or undirected, connected or disconnected even though it must be unweighted. Keynodes represent the names (ID) of nodes belonging to the graph. Our idea is to find out which and how many node can reach each keynodes within a maximum adjustable distance. Moreover for each node we provide the minimum distance from its closest keynode. The whole program has been implemented using Hadoop MapReduce in order to handle very large graphs, often used to many systems of current interest.

This implementation could be a useful model used to provide advices to users of a network. Keynodes may represent items or users of the network, this allows to find out items/user reachable within a certain distance. More keynodes reach a node, more that node can be considered interesting. The algorithm can be used in many different ways to find out the information whom you are looking for.

4. SOLUTION

The process development of the project has been split into 3 parts: The first one is the research of the ideal solution and optimization of itself; the development of the solution; and last but not least the testing, (in the order as written).

This chapter explains the first two parts (research of the ideal solution and optimization and the development solution), we talk about the testing part in the next chapter;

4.1 Investigation of the solution

The most simple (but less efficient) solution to our problem would be to use all the possible random walks (iterations n), starting for each keynode in the graph. Once finished , by comparing each path it find out the "most visited node" . This solution, however, appears to have an exponential cost in relation to the number of keynodes given as input.

To optimize the solution we try the Breadth-first search approach (BFS) using the idea that a node, once turned black (or a default color), it will never be visited. Hence we decided to implement our algorithm on a BFS structure in order to be able to use the BFS "benefits". Anyway this optimization can not be enough, in fact, in the worst-case scenario, the algorithm's cost is the same as it was before. So we implemented a convergence system in order to avoid an overload work for it. However we have to think that the algorithm was developed by using an iterative map reduce system that allow parallelism of the actions in a distributed system. In that point of view the algorithm's cost doesn't seems so expensive anymore, but we discuss about our results in the next section.

4.2 Development

We wrote the foundations thanks to "Iterative MapReduce and Counters" a map-reduce algorithm developed by users of Hadoop Wikispaces [12] since our algorithm works by using a BFS-map reduce approach. For the development we decided to use Java as programming language. We created a Java-Apache-Maven project using Apache Maven (version 3.3.3) and the Java Development Kit version 1.7. We write our project using Eclipse Mars 1 as development environment.

Before starting explain how our algorithm works we also want to improve what BFS means. Breadth-first search is a graph traversal algorithm. The search starts from the root node and the neighboring nodes are visited until there are no more possible nodes to visit. One way of performing the BFS is coloring the nodes and traversing according to the color of the nodes. There

are three possible colors for the node - white (unvisited), gray (visited) and black (finished). Before starting all the nodes are colored in white except for the source node that will be colored gray.

To develop this algorithm we need to run the same Mapper and Reducer multiple times (iterative map-reduce).Each iteration can use the previous iteration's output as its input.

"Iterative MapReduce and Counters" contains also a solution for the shortest path problem (the shortest path between two nodes can be defined as the path that has the minimum total weight of the edges along the path. if we don't talk about weighted graph the minimum weight will be just the minimum number of edges). It implements this problem using Dijkstra's algorithm.

The algorithm is designed to make it run on a graph that has only a source node (or as we prefer call it a "starter node"), instead our solution needs to has more than a single start node (i.e. the whole set of keynodes given in input). Then we modify the graph input's structure in order to be able to run the BFS starting from more than one node (those will be the keynodes). In addition we tried to adapt our input structure according to the majority of graphs we found online which allowed us to use our program in lots of graphs. So we transformed the graph input from the following format:
ID<tab>NEIGHBORS|DISTANCE_FROM_SOURCE|COLOR
to a more common format: ID<tab>NEIGHBORS.

As we can see, we haven't get colors and BFS stuff anymore, our program require just the basic structure of a graph (the node ID with the respective adjacent-node IDs). Most of the graphs found don't show the node with all the other nodes connected to it, but for each edge they write the starting and the destination node, then we developed a little algorithm to “translate” inputs as we required. Our software, to work correctly, needs another input file where will be simply written the keynodes.

Now step by step, assuming that the reader has the basic knowledge on how Hadoop MapReduce works, we are going to explain how our project works and also how it is written. First of all we want to show our program's classes and explain a little bit their functionality, after that we will explain the implementation of the software in its entirety.

4.2.1 BaseJob

Our software doesn't need a complex and structured job, so we decided to keep the Hadoop base job structure and adapt it with just slightly modifications.

The following is an abstract base class for the programs in the graph algorithm tool kit. This class provides an abstraction for setting the various classes associated with a job in a MapReduce program. The base job class contains a setupJob method that takes the job name and a JobInfo object as parameters and returns a job that is used by the driver to execute the MapReduce program [12]. In this method are set the mapper and reducer classes, the number of reducer used for the program (easily editable), the types of the output key and output value of the mapper and reducer.

Finally, into BaseJob is declared the abstract class JobInfo that contains getter methods to get the program-specific classes associated with Job.

4.2.2 Driver

This class contains the driver to execute the job and invoke the map/reduce functions and the main class. The task of the main class is just to create the output directory and call the run method.

The run method contains the driver code and it has the task of managing the entire algorithm's process. Run method, for each iteration, must perform the following tasks: checking if the process has reached the end by monitoring the iteration and if it arrived at a convergent point; getting the job configuration; setting the input/output file and their path; waiting until the job finish his work, then repeat the process again.

Driver contains other two classes, MapperCaller and ReduceCaller, which are responsible for the calling of the mapper and reducer super classes.

MapperCaller extends the MapperWorker class. MapperCaller contains a map method who takes as parameters a key (node ID), a value (node information) and the context. This method is in charge to take the node with the right parameter's value by initializing and instantiating the Node variables (Node inNode = new Node (value.toString())), then calling the map method of the super class passing the keynode, input value, Context object and the node (just created) as parameters.

ReducerCaller extends instead the ReducerWorker. ReduceCaller contains a reduce method who takes as parameters the same parameters than the map method written above. Reduce method is in charge to initializing and instantiating an empty node , then calling the reduce method of the super class passing the keynode, input value, Context object and the node (just created) as parameters.

4.2.3 Node

Node class contains the information about the node ID, the list of adjacent nodes, the distance from the source, the color of the node (color could be white for unvisited node, gray for visited node and black for the node which has been visited by all the keynodes) ,the parent node and the starter list(list of the keynodes that visited the node).

The node class, at the first iteration, read inputs for setting the nodes as required in a BFS for starting the job. In all the other iterations the node's work is reading and interpreting the node's information by the input files. Node class also contains his getter and setter method as well and a method for getting all the node's information appended in a string.

4.2.4 MapperWorker

MapperWorker is the base mapper class for the programs that use parallel breadth-first search algorithm. In this class are implemented the map method which is responsible for the mapper work.

The method takes the keynode, the input value, the Context object and Node (input node) as parameters. The map function, for each gray node, emit each of the adjacent nodes as a new node and set them. Once finished the map function writes the results (the news nodes with their setting) on the context in order to pass them at the reducer.

4.2.5 ReducerWorker

ReducerWorker is the base reducer class for the programs that use parallel breadth-first search algorithm. It combines the information for a single node. The complete list of adjacent nodes, the minimum distance from the source, the darkest color, the

parent node of the node that is being processed , and the starter list are determined in the reducer step. This information is emitted from the reducer function to the output file.

After gave you a brief introduction and a general overview about software's classes and their works, we want to follow our algorithm step by step in order to explain how it works, explaining some steps in detail.

At the beginning(first iteration) the program has to prepare input for the mapper class. So the class node read the two inputs. For each node the algorithm check if the node processed is a keynode. If it is, the node will be set as a source node (Figure 1).

```
// setting a node as starter if it is present in the keywords
if (keywords.contains(this.id)) {
    // setting the source node given in the list
    this.distance = 0;
    this.color = Color.valueOf("GRAY");
    this.parent = "source";
    this.starters.add(this.id);
```

Figure 1: setting the node information for source nodes.

If it's not a keynode, the processed node will got the setting as a "standard" node (Figure 2).

```
// setting the distance of the node
this.distance = Integer.MAX_VALUE;
// setting the color of the node
this.color = Color.valueOf("WHITE");
// setting the parent of the node
this.parent = null;
```

Figure 2: setting the node information for no-source nodes.

In the next iterations, inputs will be specifically written in such a way that we don't need to translate it anymore. In the second step each node is passed by the MapperCaller class to the mapper subclass (MapperWorker).

Mapper worker processed every gray node(called inNode), and, create (as typically in a mapper process) for each adjacent node (of the node under consideration) a new empty node(called adjacentNode) with the adjacent ID. The method also provide to setting the following adjacentNode information: the color: the color will be gray; the parent: the parent of the adjacentNode will be the inNode ID; the distance: the distance will be setting increasing the inNode distance (adjacentNode.setDistance(inNode.getDistance() + 1)); The starter list: it will be the same than inNode node (keynodes that can get closer to the parent are also able to touch the nearby node). The adjacentNode will be then written on the context, ready for be passed at the reducer.

The last work for the Mapper is to check if inNode has been visited by every keynodes, then color it black. A black node will never be visited in the whole process of the algorithm. InNode will be passed at the reducer as well. The reducer (ReducerWorked) get all the results given by the mapper (adjacentNodes and inNodes) and combines the information for a single node. In other words, when there are multiple values associated with the same node ID, only one will contain the information and will be written to the output. We think that showing the reducer code is the best way to explain how the reducer method meets those demands (Figure 3).

The comments over the code efficiently explain how the reducer works. So we would like to focus only on how the reducer gets chooses and merges the starter list for each node. The reducer combines all the starter lists with the same node ID to hold on all the keynodes that have visited the processed node. Doing this, the final list could contains duplication. For this reason the reducer method should clean the list of duplicates. Finally the list will be sorted to avoid convergences issue and write it on the output.

Reduce() at the end check if the output node needs to be visited other times and, if not, change the node's color from gray to black and save it to the context as well.

4.3 Convergence

Our algorithm uses an iterative map reduce approach, this means that all the steps listed above are iterated more times, until the program has finished the possible iterations (decided before run the algorithm) or the convergence has been achieved. So another problems comes out: how to check efficiently if we have reached the convergence (we have been dealing with huge graph). We check the convergence starting at the second iteration and the method is implemented in the Driver class.

The convergence method (compareFiles(path file a, path file b)) called by the run method , takes as parameters the last two output files (iteration, iteration-1) . Every output file is the result of union between the reducer output file parts (if the program was run with a number of reducer >1). CompareFiles() splits the two documents in smaller ByteBuffer (we decided 4 MB each part), then compare all the byte buffers to each other and return the results (Two byte buffers are confronted by comparing their sequences of remaining elements lexicographically, without regard to the starting position of each sequence within its corresponding buffer [18]).

ComparesFiles's speed is around 75 MB/second, that means that it can compute approximately 1,5GB-file in 20 second using that configuration.

```
for (Text value : values) {
    Node inNode = new Node(key.toString() + "\t" + value.toString());
    // One (and only one) copy of the node will be the fully expanded
    //version, which includes the list of adjacent nodes
    // in other cases, the mapper emits the ndoes with no adjacent nodes
    //In other words, when there are multiple values
    //associated with the key (node Id),
    //only one will
    if (inNode.getEdges().size() > 0) {
        outNode.setEdges(inNode.getEdges());}

    // Save the minimum distance
    if (inNode.getDistance() < outNode.getDistance()) {
        outNode.setDistance(inNode.getDistance());
        //if the distance gets updated then the predecessor node
        //that was responsible for this distance
        //will be the parent node
        outNode.setParent(inNode.getParent());}

    // Save the darkest color
    if (inNode.getColor().ordinal() > outNode.getColor().ordinal()) {

        outNode.setColor(inNode.getColor());}

    // save the starterlist (keywords)
    if (inNode.getStarters().size() > 0) {
         List<String> startertemp=new ArrayList<String>();
        //concatenate the two result List
         startertemp.addAll(inNode.getStarters());
         startertemp.addAll(outNode.getStarters());
         // clean and sort list for duplicates
         Collections.sort(startertemp);
         Set setPmpListArticle=new HashSet(startertemp);
        outNode.setStarters(new ArrayList <String>(setPmpListArticle));}
    }
```

Figure 3: main part of the reducer() code.

When the software finish his works, we will just have a large set of output files (depending on how many iterations the program did in that run), but we will not have the solution of our problem yet. So to find out which node/nodes has been visited by more keynodes, is sufficient to search the last output which node/nodes

has got the biggest starter-list size. In addition, about the node/nodes founded, we are able to say which node is the closest to a keynode.

5. EXPERIMENTS

In our algorithm, the main elements of the graph (ID and adjacent Nodes) are managed as a string. This makes possible to run the program on different data types graphs (i.e. a Node ID may be "1" as can be "david"). Anyway the largest part of graphs are represented by integer data types with numerical ID.

The algorithm was trained using small graphs (10 to 20 nodes with a maximum of 50 edges) so we could easily check the accuracy of the results. So we tried on small string, integer and both graph. The program worked properly and performing.

During the training part we have used directed, undirected, connected and disconnected graph, but, in this chapter we will report just the most important and significant trials. The software is giving problems just if the input graph contains explicitly a node with no neighbors (if the node is present only as a neighbor of another node, the problem does not show up). Once checked the correctness and reliability of the algorithm output on small dimensions graph, we tried to run it using bigger graphs, in order to control speed, performance and possible future optimizations.

For tests we used a HP-Pavilion-dv6 laptop 64 bit (8Gb RAM memory, CPU: Intel® Core™ i7-2670QM CPU @ 2.20GHz octa-core).We used Ubuntu 15.10 as operating system. We know that, for a map reduce problem, would be better use a distributed system for testing the whole process and the real power of a map reduce program, anyway, at the moment, we haven't got the possibility to use more computers simultaneously yet.

We took the datasets used from the SNAP Datasets Collections [19] that makes available a collection of more than 50 large network datasets from tens of thousands of nodes and edges to tens of millions of nodes and edges. In includes social networks, web graphs, road networks, internet networks, citation networks, collaboration networks, and communication networks.

We started using a sequence of snapshots of the Gnutella peer-to-peer file sharing network from August 2002. Nodes represent hosts in the Gnutella network topology and edges represent connections between the Gnutella hosts. This graphs has 8.846 Nodes and 31.839 Edges. For this graph we used four different number of input keynodes (Table 1). Keynodes values are generated randomly thanks a small java program developed by us.

Test Number	Number keynodes	Number of iteration	Time required (in seconds)
1	30	100	22
2	100	100	32
3	200	100	42
4	500	100	88

Table 1: test results using a Gnutella network with different number of keynodes.

We didn't increase the number of iterations just because the algorithm reached, every time, the convergence around the 22nd iteration.

The second graph used is a sequence of snapshots of the Gnutella peer-to-peer file sharing network too but it is bigger than the first one. In fact the graph contains 62.586 nodes and 147.892 edges. We tried the same tests with the same numbers of random-keynodes as before. The results showed a slowdown in the execution time of the algorithm. With that graph and 500 input keynodes it ran in 5 minutes time. But if we look at the large data sets present in the network nowadays like Facebook or Amazon datasets, the graph described above are not so big. Hence, the results described until now couldn't be so realistic, thinking about the use of such a algorithm in a "real" world. That means that the next graph used are pretty much "bigger". We found an internet topology graph (from trace routes run daily in 2005) with 1.696.415 Nodes and 11.095.298 Edges. The program, using 30 keynodes as input, performed in 33 minutes and in 45 minutes using 100 input keynodes (Figure 4). Using such a big graph we realize that the total time for perform the entire process of the algorithm depends on the number of input keynodes and the file size (size and complexity of the graph) given as input, but, the time to perform a single iteration, depends only on the file size taken by the processed iteration, so on the number of gray nodes that has to process as well.

We also tried to run a graph with a higher number of connections (edges), this because the whole process for a MapReduce algorithm on a graph is more affected by the number of edges than the number of nodes O(iterations*edges), but, with a less number of nodes, the single iteration time is minor.

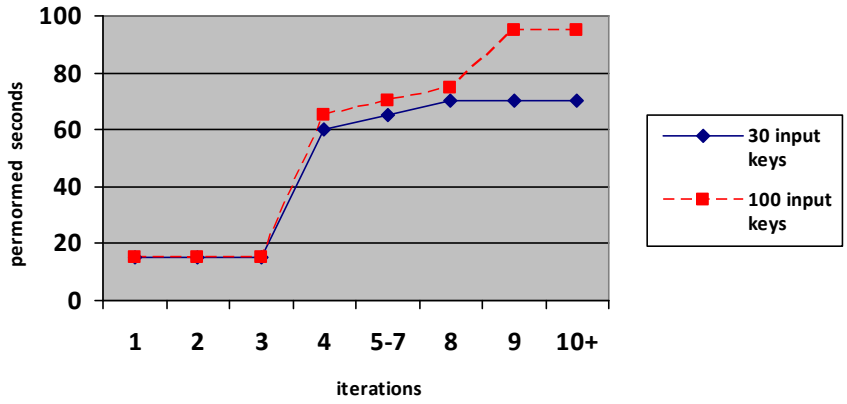

Figure 4: time used by the program to perform a MapReduce iteration with two different numbers of keywords.

The graph taken, has 3.997.962 Nodes and 34.681.189 and is come from LiveJournal Datasets. LiveJournal is a free on-line community with almost 10 million members; a significant fraction of these members are highly active. (For example, roughly 300.000 update their content in any given 24-hour period.) LiveJournal allows members to maintain journals, individual and group blogs, and it allows people to declare which other members are their friends they belong. The entire process it took four hours (using as input a set of 30 keynodes) ,fulfilling all the 100 iterations. The average iterations time for this try was about 2 minutes.

For the last test we wanted to run our algorithm on the biggest graph (Table 2) we founded. The graphs represent the Directed LiveJournal friendship social network.

Dataset statistics	
Nodes	4847571
Edges	68993773
Nodes in largest WCC	4843953 (0.999)
Edges in largest WCC	68983820 (1.000)
Nodes in largest SCC	3828682 (0.790)
Edges in largest SCC	65825429 (0.954)
Average clustering coefficient	0.2742
Number of triangles	285730264
Fraction of closed triangles	0.04266
Diameter (longest shortest path)	16
90-percentile effective diameter	6.5

Table 2: statistics about the biggest graph tested.

We couldn't see the end of the process for this test for time reasons, in fact after the 6th iteration, each iteration required 40 minutes to be completed. We planned that to complete the whole process the algorithm would have to run more than two days. By the way we have got sufficient result data for know and represent how our solution scale according to the graphs size.

6. CONCLUSION

We have built and developed this project and, by a correct formatting of the input files, it works on every types of graph without any bugs or errors. Performances, as we have seen, do not depend only on the setting and the dataset passed as input. For a good performance, users have to know how the graph is structured. The choice of the number of keynodes and iterations, that has to run, must obviously be based on the solution that we're looking for, but on the structure of data set as well. In fact, as we have been able to verify , the whole performance time of the algorithm depends on both the size of the graph and the number of edges (the higher the number of connections, the slower the convergence). The latency of each iteration, instead, is closely related to how many nodes the iteration itself has to processed and to how many keynodes are passed as input. The right number of reduces seems to be 0.95 or 1.75 * (nodes * mapred.tasktracker.tasks.maximum). At 0.95 all of the reduces can launch immediately and start transferring map outputs as the maps finish. At 1.75 the faster nodes will finish their first round of reduces and launch a second round of reduces doing a much better job of load balancing [20]. Anyway there are other issues that impact that decision (like the cluster and the number of machines that can works), but in a perfect scenario each and every reduce has to happens in parallel.

Future optimizations will be considered once tested the algorithm on a distributed system. For now we did the first and fundamental step. We've created a general and alternative way to marking suggestions in different kind of systems that can be used on a large number of different kind of situations and problems in a big data scenario.

7. REFERENCES

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Commun ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[2] M. Zaharia, A. Konwinski, A. Joseph, R. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments," presented at the OSDI, 2008.

[3] "Apache Hadoop." http://hadoop.apache.org/.

[4] "Applications powered by Hadoop." http://wiki.apache.org/hadoop/PoweredBy.

[5] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, 2010, pp. 1–10.

[6] M. Newman, "The Structure and Function of Complex Networks," *SIAM Rev.*, vol. 45, no. 2, pp. 167–256, Jan. 2003.

[7] "Breadth-first search," *Wikipedia, the free encyclopedia*.

[8] "Breath First Search - Lecture by Rashid Bin Muhammad, PhD." http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/GraphAlgor/breadthSearch.htm.

[9] K. Mehlhorn and P. Sanders, *Algorithms and Data Structures: The Basic Toolbox*. Springer Science & Business Media, 2008.

[10] "breadth-first graph search using an iterative map-reduce algorithm." http://www.johnandcailin.com/blog/cailin/breadth-first-graph-search-using-iterative-map-reduce-algorithm.

[11] "GitHub dougvk/CS462," *GitHub*. https://github.com/dougvk/CS462.

[12] "hadooptutorial - Iterative MapReduce and Counters." https://hadooptutorial.wikispaces.com/Iterative+MapReduce+and+Counters.

[13] S. Neumann, "Spark vs. Hadoop MapReduce," *Xplenty*. https://www.xplenty.com/blog/2014/11/apache-spark-vs-hadoop-mapreduce/.

[14] J. Davidson, B. Liebald, J. Liu, P. Nandy, T. Van Vleet, U. Gargi, S. Gupta, Y. He, M. Lambert, B. Livingston, and D. Sampath, "The YouTube Video Recommendation System," in *Proceedings of the Fourth ACM Conference on Recommender Systems*, New York, NY, USA, 2010, pp. 293–296.

[15] G. Linden, B. Smith, and J. York, "Amazon.com recommendations: item-to-item collaborative filtering," *IEEE Internet Comput.*, vol. 7, no. 1, pp. 76–80, Jan. 2003.

[16] L. Li, D. Wang, T. Li, D. Knox, and B. Padmanabhan, "SCENE: A Scalable Two-stage Personalized News Recommendation System," in *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval*, New York, NY, USA, 2011, pp. 125–134.

[17] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl, "Item-based Collaborative Filtering Recommendation Algorithms," in *Proceedings of the 10th International Conference on World Wide Web*, New York, NY, USA, 2001, pp. 285–295.

[18] "ByteBuffer (Java Platform SE 7)." https://docs.oracle.com/javase/7/docs/api/java/nio/ByteBuffer.html.

[19] J. Leskovec and A. Krevl, "SNAP Datasets: Large Network Dataset Collection," Jun. 2015.

[20] "GitHub paulhoule/infovore," *GitHub*. https://github.com/paulhoule/infovore.