www.ijecs.in
International Journal Of Engineering And Computer Science ISSN: 2319-7242
Volume 4 Issue 12 Dec 2015, Page No. 15183-15189

Map-based Multi-source Message Passing Model to find Betweenness Centrality using MapReduce

Nikhitha Cyril[1], *Arun Soman*[2]

[1]PG Student, Department of Information Technology,
Rajagiri School of Engineering & Technology, Kochi, India
nikhithacyril@gmail.com

[2]Assistant Professor, Department of Information Technology,
Rajagiri School of Engineering & Technology, Kochi, India
arunnediyasala@gmail.com

Abstract: ***The need for large-scale graph analysis and finding efficient methods for the same are increasing. MapReduce framework has already been accepted as a standard for large-scale analysis. Finding betweenness centrality is highly significant in finding the importance of a node in a network. In this paper, we propose a map-based method to find betweenness centrality using the multi-source message passing model. The multi-source message passing model has two phases namely the breadth-first traversal phase and the backtracking phase. In the breadth-first traversal phase, we traverse the graph in breadth-first form by sending and receiving messages and in the process find the shortest paths in the graph. In the backtracking phase, we traverse back to the source node. While doing so, we find the dependency of the source node on other vertices by transmitting messages. Both these phases are implemented in the map-based method which consists of an initial MapReduce job followed by a number of iterations of mapper functions until there are no more messages to send. We can then find betweenness centrality by summing the dependencies of all the source nodes on the vertex.***

Keywords: MapReduce, Betweenness Centrality, message passing model, map-based model.

1. Introduction

The increasing usage of web and social networks have resulted in the need for better and improved ways of graph analysis. Hadoop [11] with its MapReduce programming framework has already proved to be big help in this matter. However, most graph analysis techniques like finding betweenness centrality, pagerank, etc. involves iterative algorithms. The MapReduce framework does not support this directly. In order to make it possible multiple MapReduce jobs should be manually allocated and executed using a driver function. But, in this case the data which is mostly the same needs to be reloaded and reprocessed during each iteration thereby resulting in a wastage of I/O, network bandwidth and CPU resources. Deciding when to stop the iterations also causes an overhead. Hence, it is necessary that we find better methods to do it [1].

Twister [2], HaLoop [1], etc. are enhancements made on MapReduce to support iterative algorithms. Gupta introduced a method of graph analysis involving iterative algorithms where he separated the data that remains constant from the data that keeps varying during each iteration. This reduced the overhead caused by reloading and reprocessing the same data. Gupta's method was a map-based method that avoided the reduce phase in the iteration to save the communication time required to send intermediate results from the map phase to the reduce phase [3].

The nodes in large networks may not be equivalent. Hence, modifying a node in the network might affect the network differently based on the modified node. For example, if the node is a central node connected with a significant number of nodes is removed, then the impact it will have on the network will be much greater than that caused by an end node. Hence, it is important that the centrality of the node is measured so as to protect it from attacks and breakdowns [4].

Freeman [5] suggested three different measures for centrality namely degree, closeness and betweenness. Here, degree specifies the number of nodes connected to the node under consideration. Even though it is used to measure the node's involvement in the network, it does not consider the global structure of the network. This is where closeness comes in. Closeness centrality is the inverse sum of the shortest distances from the node to all other nodes. However, this measure is meaningless in the case of disconnected components. The third measure which is betweenness centrality is the number of shortest paths between two other nodes for which the node is a part of [6].

The betweenness centrality measure suggested by Freeman had a complexity of $\theta(n^3)$ time and $\theta(n^2)$ space. Also, it could only be used in unweighted graphs. Brandes [7] introduced an efficient and faster method which enabled finding betweenness centrality in weighted graphs as well. He considered the fact that transaction might be quicker in large number of strongly connected nodes than in a lesser number of weakly connected nodes as the strongly connected nodes will have frequent contact than the weakly connected nodes [6].This technique had a space complexity of $O(n + m)$ and a time complexity of $O(nm)$ and $O(nm + n^2 \log n)$ for unweighted and weighted networks respectively. Zeng [8] introduced a multi-source message passing model to implement Brandes technique on the MapReduce framework. In this method the vertices can send and receive messages from other vertices during each iteration. The algorithm consists of a breadth- first traversal phase to find the shortest paths and a backtracking phase to find the dependency of the source vertex on other vertices. Finally, the dependencies of all the source nodes on the vertex are summed to find the betweenness centrality of that vertex.

In this paper, we introduce an algorithm to implement Zeng's multi-source message passing model to find the betweenness centrality in the map-based form suggested by Gupta. The breadth-first traversal and backtracking phases in this method will have an initial MapReduce job to send the first set of messages followed by iterations of Mapper functions to receive the messages and to send the next set of messages until there are no more messages to send. After the execution of both these phases, the betweenness centrality is calculated by summing the dependencies of source vertices.

The rest of the paper is structured as follows. Section 2 describes the multi-source message passing model and how to find betweenness centrality using the model. Section 3 describes the proposed method which is finding betweenness centrality using the map-based multi-source message passing model. Section 4 evaluates the performance of the proposed method and compares it with the multi-source message passing model. Finally, we arrive at a conclusion.

2. Multi-source Message Passing Model

The message passing model is executed on a direct graph where each vertex of the graph stores the vertex details and has a message table to store the messages that it receives. The vertex details include the state of the vertex. The vertex state is active only when it has to send messages. The vertex state becomes inactive when it has no more messages to send. The inactive vertices do not take part in the subsequent iterations unless they receive one or more messages and are activated again. During each iteration, the active vertices will send the active messages in its message table to all its adjacent vertices and become inactive. When there are no more active vertices, the algorithm terminates [8].

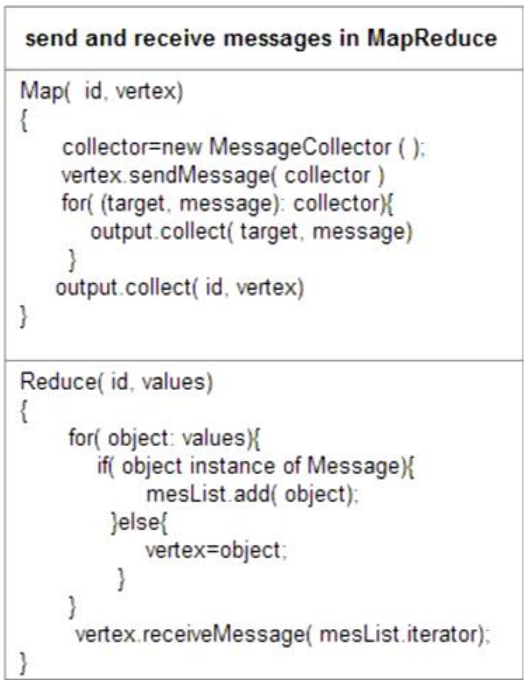
The message passing model can be used to find, the shortest distance, betweenness centrality, for enumerating triangles, etc. In case of single-source message passing model, there is only one active node in the beginning. For example, while finding the shortest distance, we start from the source node as active. In this case we can only find the shortest path from the given source node. However, in cases where we need to find the shortest path from more than one source node we might have to repeatedly execute the algorithm with all the source nodes. This will not only result in more number of iterations but also in the iterations not being fully used as there will be many inactive vertices in each iteration [8].

Hence, we use multi-source message passing model where we can have more than one active vertex in the beginning. To enable this we need the message to contain details like the source vertex and state of the message along with the actual message content in order to distinguish the message from the messages send from other source nodes [8].

2.1 Implementing the Message Passing Model on MapReduce

The MapReduce method consists of the Map phase for mapping data into <key, value> pairs and Reduce phase to obtain an aggregated value for each key generated in the map phase [9], [10]. Each vertex of the graph will contain details like vertex id, list of adjacent vertices and the message table. In addition, the vertex will also contain two functions named sendMessage() and receiveMessage() for sending and receiving messages respectively. These functions are defined based on the purpose of the algorithm [8].

The map function will call the sendMessage() to obtain the list of active messages that are to be send by the vertex. Since these messages are to be send to each of the vertex's adjacent vertices, the map function will output <adjacent vertex id$_i$, list of active messages> as the<key, value> pair where $0 \leq i \leq$ number of adjacent vertices. The map function will also output <vertex id, vertex details> along with the above [8].



```
send and receive messages in MapReduce

Map( id, vertex)
{
    collector=new MessageCollector ( );
    vertex.sendMessage( collector )
    for( (target, message): collector){
        output.collect( target, message)
    }
   output.collect( id, vertex)
}

Reduce( id, values)
{
     for( object: values){
        if( object instance of Message){
            mesList.add( object);
        }else{
            vertex=object;
          }
     }
     vertex.receiveMessage( mesList.iterator);
}
```

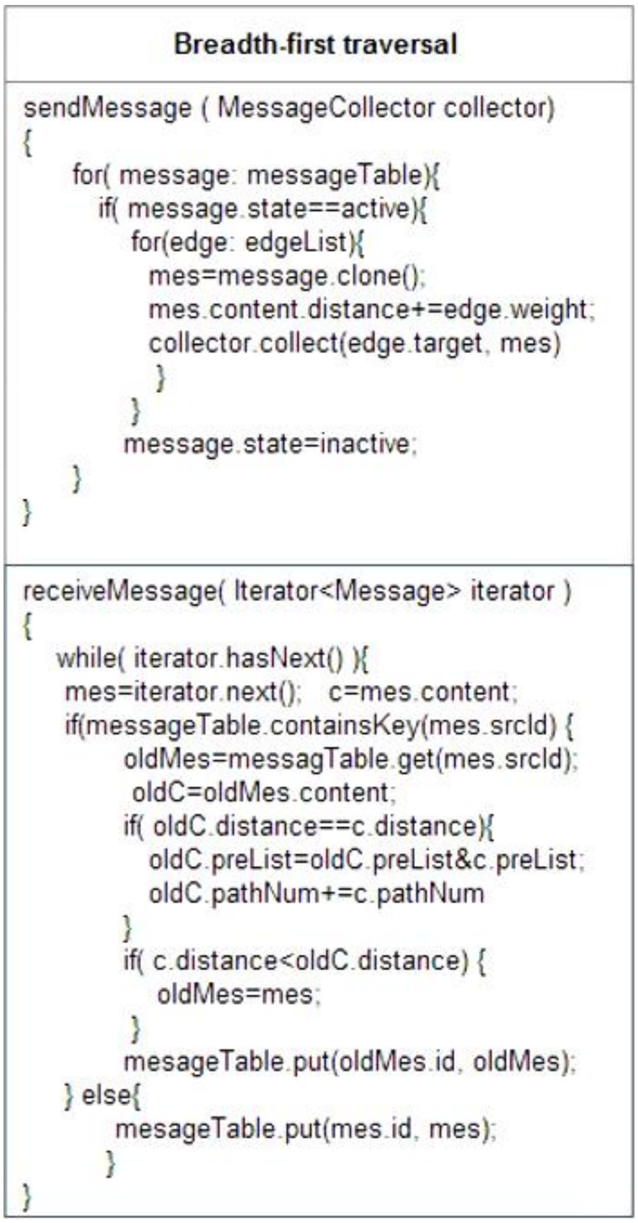
Figure 1: Implementing Message Passing model using MapReduce [8].

The reduce function will have a vertex details and the lists of received messages as values for each vertex. If the value is vertex details, then create a vertex using those details. Make a collection of all the other values which are messages to be received and then call the receiveMessage() of the created vertex to add those messages to the message table of the vertex. The output of the reduce function will be this vertex. Figure 1 shows the MapReduce algorithm explained here [8].

2.2 Finding Betweenness Centrality

Finding the betweenness centrality using multi-source message passing model consists of two phases. The first phase is the breadth-first traversal phase where we find the shortest distance from the source node to the current node as well as the number of shortest paths between them during the traversal. The second phase is a backtracking phase where we find the dependency of the source node on the current node while backtracking [8].

Each vertex will contain the vertex id, list of adjacent vertices and the message table. Each message in the message table will contain the state of the message, predecessor list, distance from the source vertex, number of shortest paths and the dependency of the source vertex on the current vertex. If the state of the message is 0, then the message is inactive and if it is 1, then the message is active. If the message table of a vertex contains active messages, then the vertex is active and it will send the active messages to the adjacent vertices. Initially, each vertex will contain an active message of the form (<1, vertex id, 0, 1, 0.0) [8].



```
Breadth-first traversal

sendMessage ( MessageCollector collector)
{
    for( message: messageTable){
      if( message.state==active){
         for(edge: edgeList){
            mes=message.clone();
            mes.content.distance+=edge.weight;
            collector.collect(edge.target, mes)
            }
         }
         message.state=inactive;
    }
}

receiveMessage( Iterator<Message> iterator )
{
   while( iterator.hasNext() ){
    mes=iterator.next();   c=mes.content;
    if(messageTable.containsKey(mes.srcId) {
         oldMes=messagTable.get(mes.srcId);
          oldC=oldMes.content;
         if( oldC.distance==c.distance){
            oldC.preList=oldC.preList&c.preList;
            oldC.pathNum+=c.pathNum
         }
         if( c.distance<oldC.distance) {
             oldMes=mes;
          }
         mesageTable.put(oldMes.id, oldMes);
    } else{
        mesageTable.put(mes.id, mes);
        }
}
```

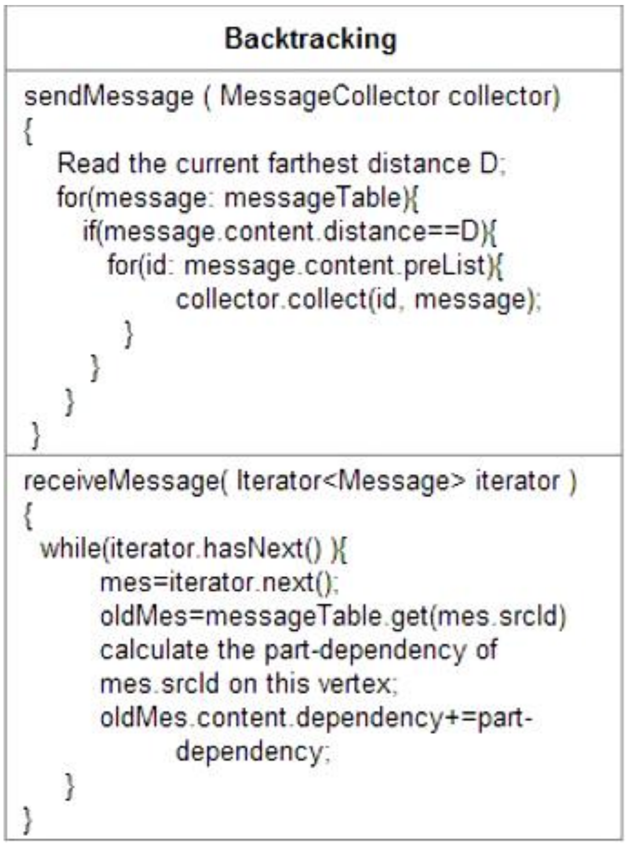
Figure 2: Algorithm for Breadth First Traversal [8].

Figure 2 shows the algorithm for the sendMessage() and receiveMessage() for the breadth first traversal. The sendMessage will return the list of active messages with the modified distance considering the weight of the edge from the vertex to the adjacent vertex. The receiveMessage() first checks if the vertex has already received a message from the same source. If it has, then it will compare the distance of the old message with that of the new message. If the distance of the old message is less than the new message, then it will ignore the new message and if the distance of the new message is lesser, then it will replace the old message with the new message. If both the messages have the same distance, then the path of the new message will be stored along with the path of the old message and the number of paths of old message will be increased by the number of paths of the new message. If there are no messages from the same source id, then the message will be stored in the message table [8].

In the backtracking phase we start from the vertex that is farthest from the source node. This will be the vertex with the largest distance. Then we traverse back to the source vertex and in the process we find the dependency of the source node on the vertex. The dependency of the source vertex, s, on any vertex, v is defined by the formula given below.

$$\delta_s(v) = \sum_{v \in p(\omega)} \frac{\theta_{sv}}{\theta_{s\omega}}(1+\delta_s(\omega)) \qquad (1)$$

Here, θ_{sv} and $\theta_{s\omega}$ is the number of shortest paths between s and v, and s and ω respectively, $p(\omega)$ is the predecessor list of vertex ω [8].



```
Backtracking

sendMessage ( MessageCollector collector)
{
   Read the current farthest distance D;
   for(message: messageTable){
     if(message.content.distance==D){
       for(id: message.content.preList){
              collector.collect(id, message);
         }
       }
     }
 }

receiveMessage( Iterator<Message> iterator )
{
  while(iterator.hasNext() ){
        mes=iterator.next();
        oldMes=messageTable.get(mes.srcId)
        calculate the part-dependency of
        mes.srcId on this vertex;
        oldMes.content.dependency+=part-
               dependency;
    }
}
```

Figure 3: Algorithm for Backtracking [8].

In the sendMessage() of backtracking phase, we have to find the current farthest distance, D. The messages with D as the distance are collected and send to the predecessor nodes. The receive message finds the part dependency of the vertex from which it receives the message using the above formula (i.e., $\frac{\theta_{sv}}{\theta_{s\omega}}(1+\delta_s(\omega))$ where ω is the vertex from which it received the message) and adds it to the dependency of the message. The backtracking phase terminates when D becomes 0. The algorithm for the sendMessages() and receiveMessages() are shown in figure 3.

Once the breadth-first traversal and backtracking is completed we can find the betweenness centrality of the vertex by adding the dependency of all the source vertices on that vertex [8].

Figure 4 shows an example for finding betweenness centrality of a graph with five vertices. Figure 4(a) shows each iteration of the breadth-first traversal and 4(b) shows each iteration of the backtracking phase. The active messages are shown in bold [8].

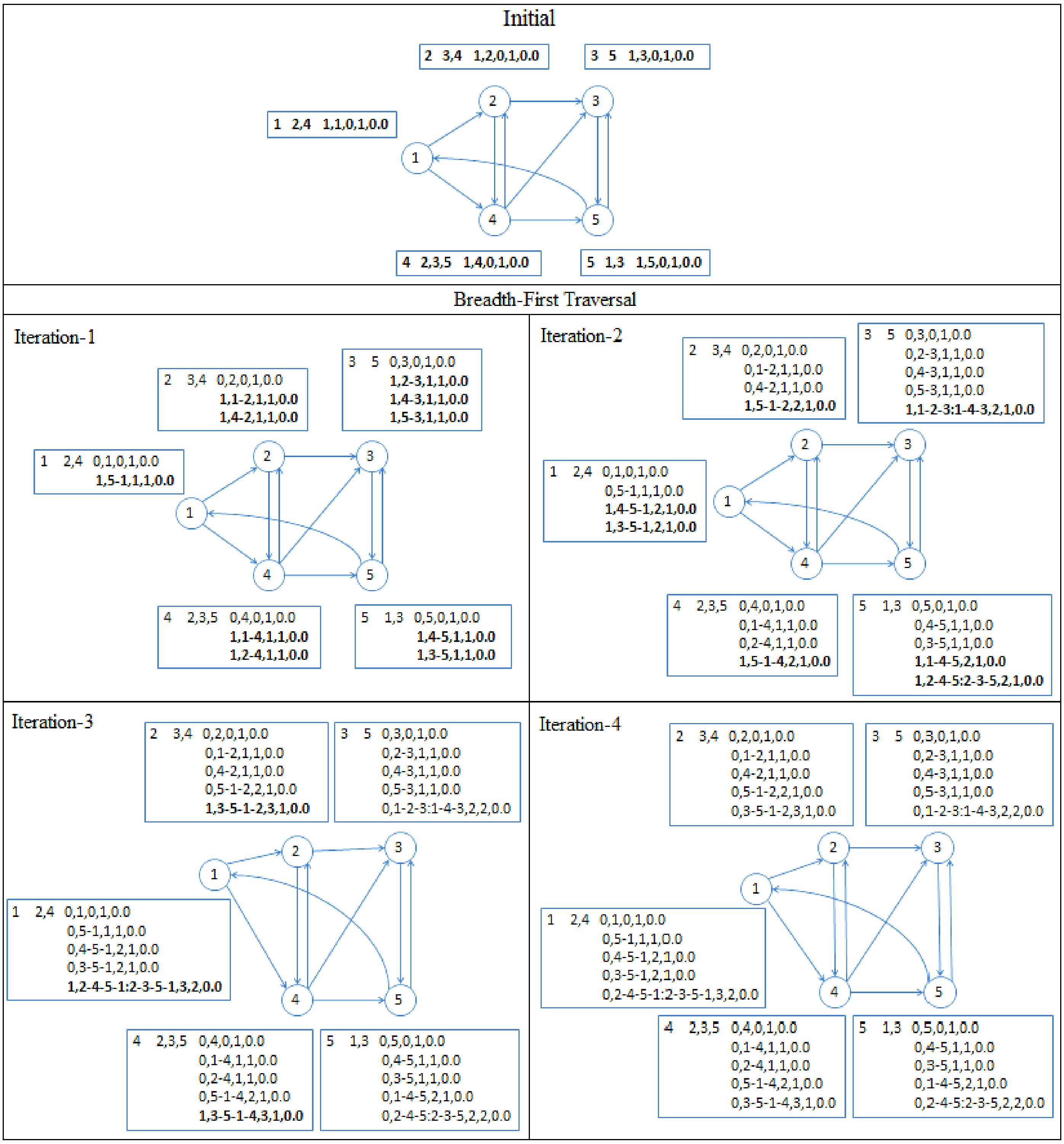


Figure 4(a): Example showing the iterations of breadth-first traversal

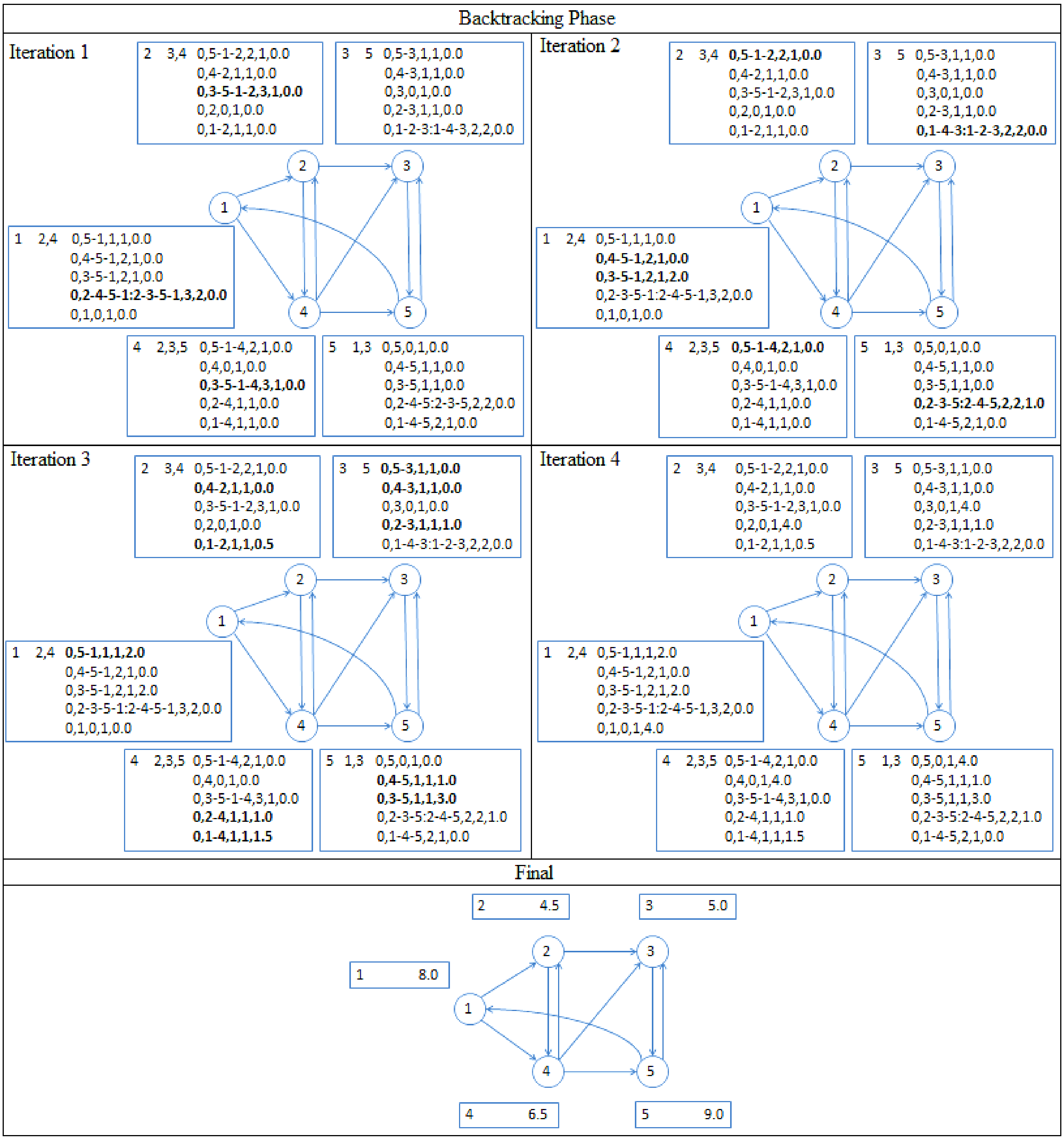


Figure 4(a): Example showing the iterations of backtracking phase. The final image shows the betweenness centrality obtained.

3. Proposed Method

Gupta [3] introduced the method of Map-Based Graph analysis on MapReduce. Through this method, he separated the graph topology which remains constant throughout the iterations from the intermediate details obtained during the iterations. A parallel merge-join is performed at each map stage to combine these details and thereby the communication time is reduced considerably. The communication time reduced further by eluding the reduce phase of MapReduce and thus saving the communication time between the map and reduce phases.

The proposed method involves converting the multi-source message passing model into the map-based form to find the betweenness centrality more efficiently. Also, once a vertex receives all the messages, then it is possible to send the next set of active messages from that vertex in the same iteration i.e., it doesn't have to wait till the next iteration to send those messages. Hence, in each map function, it will first receive the messages and then send the next set of active messages.

This method requires an initial MapReduce phase to send the first set of messages and also to separate graph topology from the intermediate details which in this case is the messages transmitted and the message table. These messages are stored in the form of multiple output files each of which acts as a graph partition and the graph topology is stored as the normal output of the MapReduce.

The figure 5 shows the algorithm for the initial MapReduce task. The input is the list of edges in the graph i.e., from and to vertices of the edge. The map function outputs these from and to vertices as output. Thus, in the reduce function will have for each key, the set of edges as the values. These data are used to form the vertex. It then calls the vertex's sendMessage() to get the list of active messages. For each edge of the vertex, these messages are stored to a file named as the edge's id. The reduce function also stores the message table to a file with the name as the vertex's id. The list of edges of the vertex is given as the normal output of the function and will serve as the input to all the following iterations of map-based MapReduce jobs.

```
Initial Map and Reduce algorithm
procedure MAP(from,to)
    output(from,to)
end procedure
procedure REDUCE(key,values)
    vertex.id=key;
    vertex.edges=values;
    msgList=vertex.sendMessage();
    for (edge: vertex.edges) do
        write to graph partition named edge(edge, msgList)
    end for
    write to graph partition named key(key, vertex.msgTable)
    write(key,vertex.edges)
end procedure
```

Figure 5: Initial MapReduce algorithm

The initial MapReduce job is followed by a number of iterations of the map-only jobs which receives and sends messages to necessary for finding the betweenness centrality. Each mapper initially performs a merge-join between the mapper input which is the vertex and edge list and the graph partitions containing the message table and send message list. During the join, the mapper initially aggregates the graph partitions. The messages from the message table are stored to a TreeMap named map and the send message list is stored to a dictionary D. Now, map will contain the message table of all the vertices and D will contain all the message list send to each vertex. The input to the map function is the vertex id and its edge list. This along with the message table for that vertex from map will form a vertex. The mapper calls the receiveMessage() of that vertex to add the message list from D to the vertex's message table. It then calls the sendMessage() to send the new active messages to its edges. This message list and the message table of the vertex are stored to a sequential file that forms the new graph partitions that is used for the next iteration. Obviously, there are no reduce functions for these MapReduce jobs. The algorithm for these mapper functions are shown in figure 6.

The sendMessages() and receiveMessages() used in the proposed method are same as those used in the multi-source message passing model for finding betweenness centrality. The initial MapReduce job and the iterative map-based jobs should be implemented for the breadth-first traversal and the backtracking phase using the sendMessages() and receiveMessages() for the same(algorithms for which are shown in figure 2 and 3 respectively).

Once the breadth-first traversal and the backtracking phases are completed, we can find the betweenness centrality of all the vertices by adding the dependency of all the source vertices on that vertex.

```
Mapper functions to send and receive messages
procedure INITIALIZE
    msg=Read Graph Partition
    if (msg is active) then
        D.put(key,msg)
    else
        map.put(key,msg)
    end if
end procedure
procedure MAP(key, value)
    vertex.id=key;
    vertex.edges=value;
    vertex.msgTable=map.get(key);
    vertex.receiveMessages(D.get(key));
    if vertex has active nodes then
        msgList=vertex.sendMessage();
        for (edge: vertex.edges) do
            write to graph partition named edge(edge, msgList)
        end for
    end if
    write to graph partition named key(key, vertex.msgTable)
end procedure
```

Figure 6: Algorithm for the Mapper functions

4. Experiment Result

The experiment was performed on a hadoop cluster set up using the Amazon EC2 Servers. Ubuntu Server 14.04 LTS was used. Hadoop 1.2.1 was installed on each server and the multinode setup was formed. The master node was used extensively as the Namenode and JobTracker. The 3 worker nodes acted as the DataNodes and TaskTrakers. One of the worker node was also set as the Secondary NameNode. The input dataset used is a graph topology consisting of 100 nodes.

The graphs below show the comparison between the multi-source message passing model and the map-based multi-source message passing model. Figure 7 shows the time taken for each of the iterations during the breadth-first traversal and Figure 8 shows the time taken for each of the iterations of the backtracking phase. Both the graph shows that the each iteration of the map-based method took much less time than that of the multi-source message passing model.

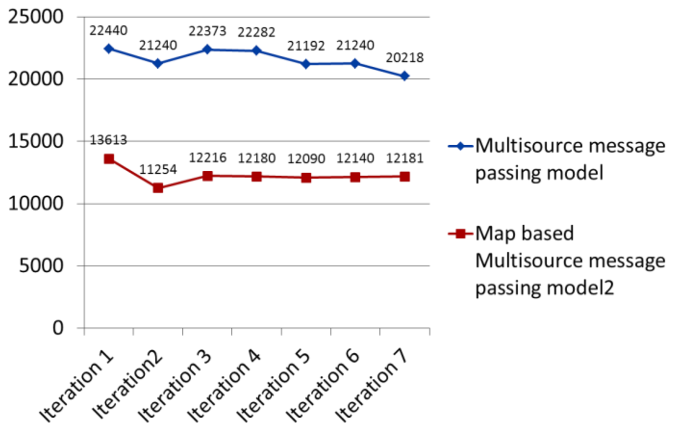


Figure 7: Comparison between the time taken during each iteration of breadth-first traversal

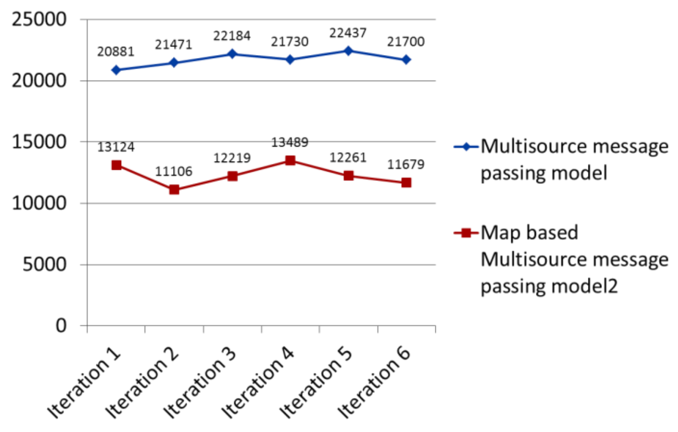


Figure 7: Comparison between the time taken during each iteration of backtracking phase

For multi-source message passing model, the average time taken for an iteration of Breadth First Traversal was 21,569ms and the average time taken for an iteration of Back Tracking was 21,733ms. Total Time taken for finding betweenness centrality using this method was 3,23,078ms. For Map-based multi-source message passing model, the average time taken for an iteration of Breadth First Traversal was 12,239ms and the average time taken for an iteration of Backtracking was 12,313ms. Total Time taken for finding betweenness centrality using this method was 2,24,323ms. The figure 8 shows this chart in a logarithmic scale.

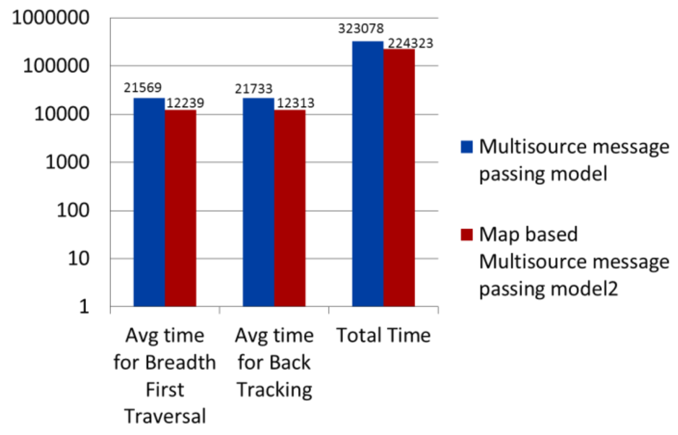


Figure 8: Comparison between the multi-source message passing model and the map-based multi-source message passing model

From these statistics, it is clear that each iteration of the map-based multi-source message passing model takes much less time than normal multi-source message passing model both during the breadth-first traversal as well as the backtracking phase. Even though the initial MapReduce job for the breadth-first traversal and backtracking phase results in an extra overhead, this overhead is proved to negligible compared to the total time saved in the Mapper functions.

5. Conclusion

Betweenness Centrality is a significant measure in finding the importance of a node in a network. However, the existing methods to find betweenness centrality are time consuming iterative methods with high complexity. This paper proposes a map-based multi-source message passing model that is faster and more efficient than the existing techniques. Experimental studies were conducted to compare the map-based method with the multi-source message passing model for finding betweenness centrality and it was found that the proposed method was about 44% faster than the multi-source message passing model.

References

[1] Y. Bu, B. Howe, M. Balazinska, M. D. Ernst, "HaLoop: efficient iterative data processing on large clusters," In Proceedings of the VLDB Endowment, 3(1-2), pp. 285-296, 2010.

[2] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S. H. Bae, J. Qiu, G. Fox, "Twister: a runtime for iterative mapreduce," In Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, pp. 810-818, 2010.

[3] U. Gupta, L. Fegaras, "Map-based graph analysis on MapReduce," In 2013 IEEE International Conference on Big Data, pp. 24-30, 2013.

[4] M. Barthelemy, "Betweenness centrality in large complex networks," The European Physical Journal B-Condensed Matter and Complex Systems, 38(2), pp. 163-168, 2004.

[5] L. C. Freeman, "Centrality in social networks conceptual clarification," Social networks, 1(3), pp. 215-239, 1979.

[6] T. Opsahl, F. Agneessens, J. Skvoretz, " Node centrality in weighted networks: Generalizing degree and shortest paths," Social Networks, 32 (3), pp. 245-251, 2010.

[7] U. Brandes, "A faster algorithm for betweenness centrality*," Journal of Mathematical Sociology, 25(2), pp. 163-177, 2001.

[8] Z. F. Zeng, B. Wu, T.T. Zhang, "A multi-source message passing model to improve the parallelism efficiency of graph mining on MapReduce," In 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), pp. 2019-2025,2012.

[9] "MapReduce Tutorial," (n.d.), Hadoop.apache.org., June 29, 2015. [Online]. Available: https://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html. [Accessed: Nov. 21, 2015].

[10] R. Ho, "How Hadoop Map/Reduce works," Dzone.com, Dec. 16, 2008. [Online]. Available: https://dzone.com/articles/how-hadoop-mapreduce-works. [Accessed: Nov. 21, 2015].

[11] T. White, Hadoop: The definitive guide, O'Reilly Media, Inc., United States of America, 2012.