Scaling graphs with Giraph – Finding Community Structural hole

Pramod Chavan
College of Information Studies
Master Information Management
University of Maryland, College Park
pchavan@umd.edu

Sarika S. Hegde
College of Information Studies
Master Information Management
University of Maryland, College Park
Sarika1@umd.edu

Namesh Kher
College of Information Studies
Master Information Management
University of Maryland, College Park
nkher@umd.edu

ABSTRACT
In real world, there are communities that hold certain group of nodes together. Say when community A wants to pass some information to community B, there has to be some common link that connects these communities together, without which it would be difficult to relay information from one group to another. To think from a social network point of view, there are some common edges that connect two small communities in a wider network, which can be source to information diffusion. This paper comes up with an algorithm, based on the Girvan-Newman method for Community Detection [2], to detect edges with 'high betweenness' connecting the communities depending on their betweeness centrality. The first part of the algorithm is to perform a **M**ulti-**S**ource **B**readth **F**irst **S**earch (MSBFS) for each node as a source and create a corresponding parent array. The second part of algorithm uses the generated parent array to increment the counts for the edges that appear in the shortest path for each source to destination node combination. The result is a count of all the edges in the graph with their counts in decreasing order. The nodes connecting the edge with highest value represent the structural hole in our network.

1. INTRODUCTION

A graph is a set of vertices and links that connect them. Studying the properties of a graph relates to the fields on mathematics and computer science. To define a graph more formally, a graph *G* can be defined as an ordered pair that consists of a set of vertices *V* and a set of Edges *E*. Therefore *G (V, E)* consists of *{V}* and *{E}*. Each edge e that belongs to *E* in the graph connects two vertices u, v belonging to V, which is the endpoint of the e.

When thinking of graphs in the real world, we can find networks in various disciplines like online social networks, the World Wide Web, transportation network and many more. For any typical real world network, there exists few individual who servers as intermediaries between connecting cluster. Through connecting bridges between communities, such individual's acts as important brokers exchanging information between not directly connected nodes.

Our algorithm, works on determining the structural holes within a given social network. When two clusters within a graph have no redundant information then one can say there is a structural hole between them. Bridging this gap within the network can help pass information within the two clusters. And one can say that the common edge between the two is one with access to information from diverse sources and clusters.

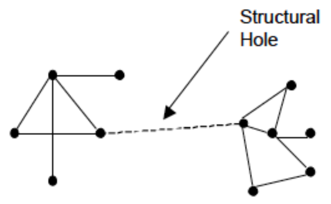


Figure 1. A Structural Hole in an organization depicting the role of an HR manager. Say any of the employees has to reach to the higher authorities in the organization; one can get in touch with HR who redirects the information to the apt person. Thus, HR servers as an important entity bridging gap between employee and management network.

2. RELATED WORK AND MOTIVATION

A lot work has been done on detecting community structures within a complex network. One of the known approaches is the '*The Girvan-Newman'* [2] algorithm that is used for detecting communities in a network. It is a divisive algorithm used on a graph that is not a tree structure or in which there is more than one path to a vertex. A community may be a group of people with similar interests. However, the weak connections within the network [Figure 2] may be the one who possess the highest degree of information.

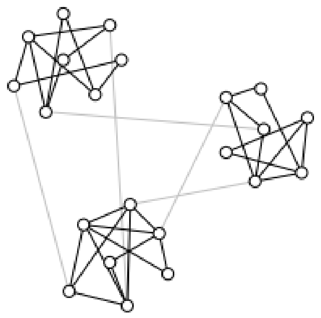


Figure 2. A schematic representation of a network within a community structure. In this there are three communities of densely connected vertices with lower density of connections [2]

Girvan Newman Algorithm can be described in following way [1]:

1) Calculate edge betweenness for every edge in the graph.

2) Remove the edge with highest edge betweenness.
3) Calculate edge betweenness for remaining edges.
4) Repeat steps 2-4 until all edges are removed.

While the above-mentioned research work focus on deriving communities from an immensely large social network, we instead use a similar model to get the structural holes in an online network. We developed a divisive method, which builds on the existing algorithm of calculating *edge betweenness by* obtaining edge weight (number of shortest path passing through the edge endpoints) for all edges.

3. TECHNOLOGY USED

Implementing graph algorithms are always challenging especially when the graph size is huge. A lot of things like fault tolerance; synchronization, memory, scalability and processing power have to be considered before choosing a particular technology for usage. While Map Reduce [6] seems a reasonable choice when it comes to processing Big Data [7, 8] it brings with itself a set of problems when it comes to mining and studying graphs. The main reason for this is the in-built characteristics of graphs that they involve heavy computations amongst and across vertices, which involve passing messages along the edges. Techniques that (Lin and Dyer) present in like Range partitioning the graph, using combiners, applying in mapper combining, and using the Schimmy algorithm prove to address these issues with grace however are challenging to implement. Based on the Map Reduce framework, Apache Giraph is a technology that seems promising when it comes to processing graphs.

Apache Giraph is an iterative graph processing system which is an open source counterpart to *Pregel* [4] which is the graph processing architecture developed at Google in 2010.Giraph is based on *Bulk Synchronous Parallel* [3, 10] developed by Leslie Valiant. Giraph adds features as compared to Pregel like master computation, edge oriented input, out of core computation and many more into its architecture.

As a part of our work we explore the architecture of Giraph to implement Multi-Source Breadth First search and call it ***Giraphed Multi Source BFS***. To get a high level overview of how Giraph works is it accepts an input graph in the form of (Vertex, Set of Edges) or Edges (Pair of Nodes) and runs a sequence of iterations, which are also called *Superstep*s. During a *Superstep* the framework executes a user-defined function called *compute ()* which runs over each vertex in a single superstep. It also allows a vertex to send messages to other vertices in the graph. A user adds logic to the compute method. A vertex can vote to halt at any iteration, which signifies that the vertex has stopped running from the next iteration onwards. However if a vertex receives messages from others, it again becomes active for the next superstep. The algorithm stops when all supersteps are done executing or when all vertices vote to halt. The following four steps outline the basic framework:

a. Superstep barriers
b. Sending and receiving messages from neighbors
c. Update the edge value
d. Vote to halt or wake up the inactive vertex.

Example in figure 3 shows a pictorial representation of how computation takes place through message passing at each step in Giraph.

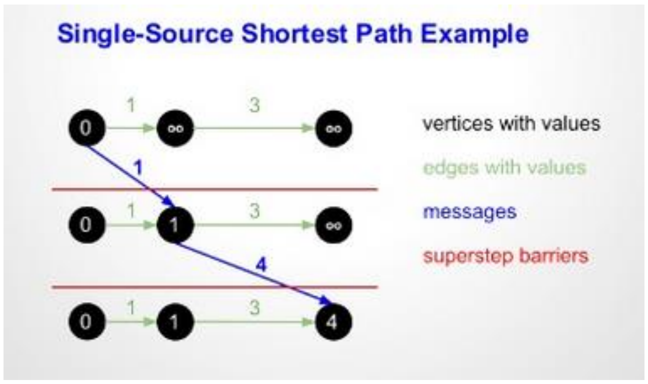


Figure 3. Diagrammatic representation of how Giraph performs computation at each vertex in the graph at each super step.

Essentially one can think of Giraph as a vertex centric processing approach. You have to think like a vertex when designing an algorithm in Giraph.

4. DATA CLEANING AND TRANSFORMATION

Having tested the data on tiny graph text file we tried to explore the scalability of the *Giraph* by using email-Enron dataset [11], a graph with 36692 nodes and 183831 edges. For the purpose of memory management, we work on a sample of the graph. To generate the graph sample randomly we came up with the following algorithm.

Algorithm: Sampling of the Graph

```
1. class Sampling
2. random_NodeId <- RandomGenerator.nextInt
   (Num_of_nodes)
3. Edge <- Queue
4. If the vertex id is not already visited,
   a. Remove from the queue
   b. Perform step 1, 2 3 for the same
5. Continue Steps 1 to 4 till queue is empty.
```

5. ALGORITHM

The algorithm to calculate the structural hole is done using the following two steps:

a) We work on performing Multi-source BFS on the input graph to create a *Parent Array*.

b) In the second half of the algorithm we traverse backwards on the created array to find the edges and the number of shortest paths passing through them.

c) The edge with the highest count is the structural hole that connects graph communities.

Explanation with an example:

Input Graph:

Format - [Source_id, source_value,[[dest_id,edge_value],]…]

[0,0, [[1,1], [3,3]]]

[1,0, [[0,1], [2,2], [3,1]]]

[2,0, [[1,2], [4,4,]]]

[3,0, [[0,3], [1,1], [4,4,]]]

[4,0, [[3,4], [2,4]]]

a) Multi Source Breath First Search

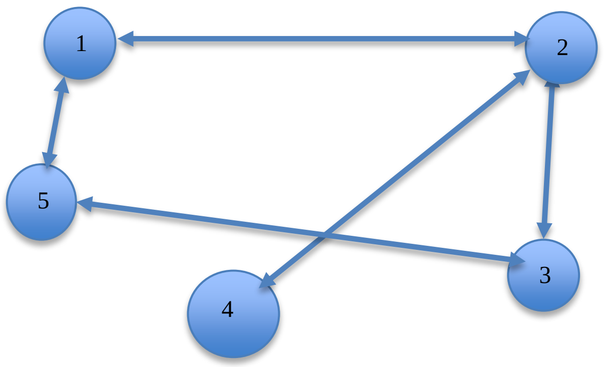


Figure 4a. Input Graph

Step 0: Each node will act as source and send messages to every connecting edge

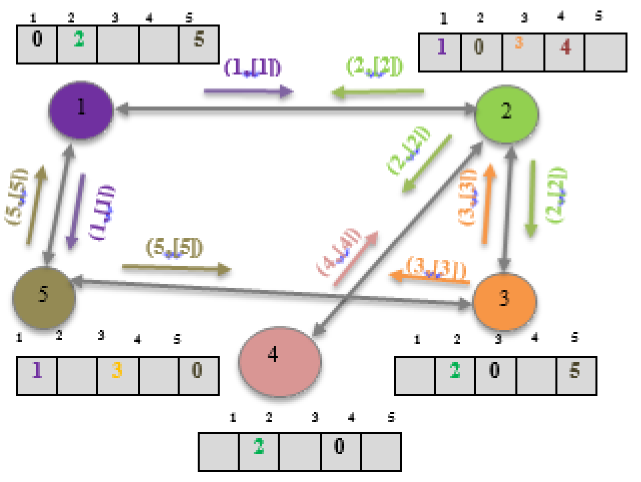


Figure 4b. ***Super Step 0***

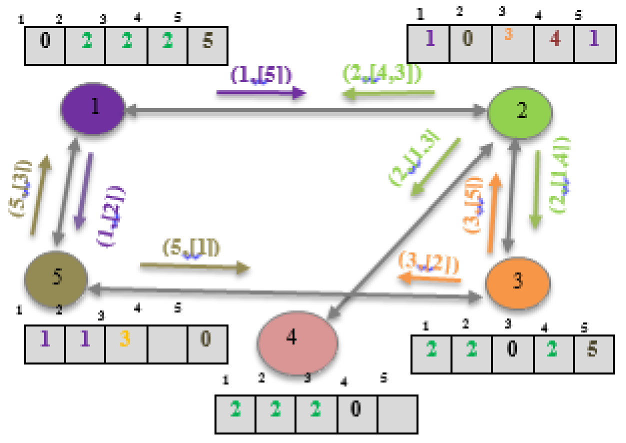


Figure 4c. ***Super Step 1***

\

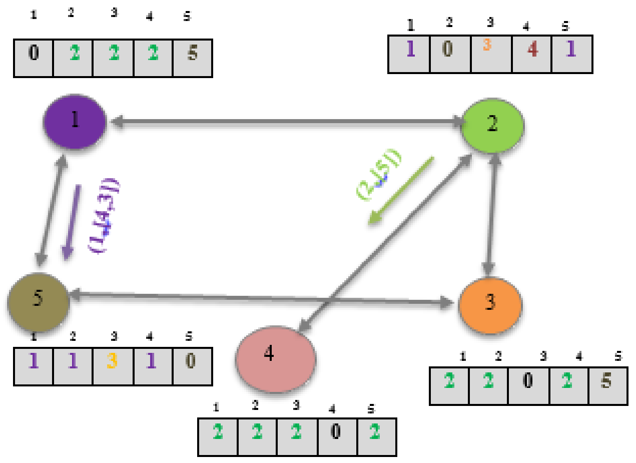


Figure 4d. ***Super Step 2***

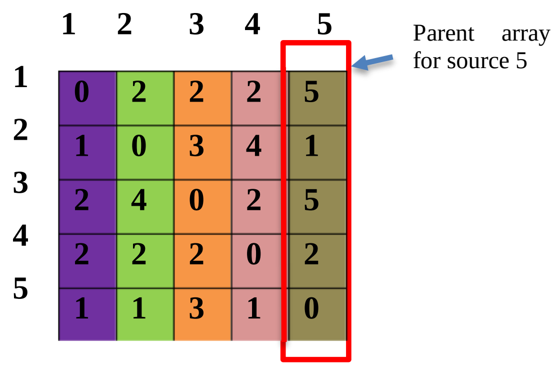
For superstep 0, every node is active and sends message in below format to all its edges (Fig 4b)

Message: [ParentId, [SourceID]] => [VertexId, [VertexID]]

For each other superstep, for every message received by a node, the node gets the SourceId's to go to the vertex array index and updates its value with received parent id.(Fig 4c,4d)

For eg: If node 1 receives message (2, [4, 3]) it updates the vertex array at index 4 (Source Id) and 3(Source Id) with value 2 (Parent ID).

After all vertex arrays get filled and superstep equivalent to graph diameter reaches, the computation is halted. Parent array for each vertex as source is formed can be retrieved from all vertex arrays as below:



	1	2	3	4	5
1	0	2	2	2	5
2	1	0	3	4	1
3	2	4	0	2	5
4	2	2	2	0	2
5	1	1	3	1	0

Parent array for source 5

Figure 5. *Parent Array Lists*

b) Back Traversal

Using parent array for each source, shortest path from that source to every other node in the graph can be obtained. Shortest path from each node to its source node is traversed incrementing its edge value by 1 along its way.

As parent array for a source node is distributed among all nodes, all algorithms work as below:

For superstep 0, each node reads its vertex array to form message to be sent its edges. The value to be sent to a edge is the index value corresponding to the sending neighbor. For every message sent, the corresponding edge value is incremented by 1.

For eg: Vertex 1 sends it neighbor 2 values 2, 3, 4 (index's for value 2 from vertex array) while it's other neighbor 5 value 5 (index for value 5 from vertex array).

In other superstep, for every message received at a node, if parent id in message is not equal to its vertex id, the message is forwarded to correct edges. The edge to which the message is to be forwarded is determined using vertex array. The received parent id is searched in the array and the corresponding index is the edge to which message needs to be forwarded. For every message sent, the corresponding edge value is incremented by 1.

When processing of superstep equivalent to diameter of graph is reached the computation is exhausted. All edge values are then scanned to determine the edge with highest value that connects the structural holes.

Algorithm: *Multi- source Breadth First Search*

```
class MultiSourceBreadthFirstSearch
    method pre_superstep
        If SUPER_STEP = 0
            Set MAX_SUPERSTEPS
    ------------Multi – Source BFS ---------------
    method Compute
        If  SUPER_STEP == MAX_SUPERSTEPS
            Vertex VOTE TO HALT
        If SUPER_STEP = 0
            Source_Id = Vertex Id
            Vertices = Total_Number_Vertices
            LIST <- new ARRAYLIST of Vertices
            LIST.ADD(0);
            S_IDS <- new ARRAYLIST of Source_Id
            S_IDS.ADD(Source_Id)
        SEND_MESSAGE_TO_ALL_EDGES
            Vertex VOTE TO HALT
        ELSE
            Current_Vertex_Id = Vertex ID
    CURRENT_LIST<- new ARRAYLIST Vertex_Value
    TO_SEND <- new ARRAYLIST of Source Messages
    SET  <- new HASH SET to avoid Duplicate messages
    for all  edges : vertex.getEdges do
       current_Edge = get Target Vertex Id
          For all messages do          //iterate over Messages
             Get Parent_Id from Message
             SOURCE_IDS = list of sources from Message
          for all SOURCE_IDS do   // Traverse on source list
             If Current_Vertex_Id! =  SOURCE_IDS (i)
             && SOURCE_IDS.get (SOURCE_IDS) == 0
                 SOURCE_IDS. SET
                     (SOURCE_IDS.get (i), Parent_Id)
                 If Parent_Id! = current_Edge
                   for all SOURCE_IDS do
                   SET.ADD (SOURCE_IDS.get (i))
          for all SET  do
               TO_SEND.add(SET)
    SEND_MESSAGE
    (edges.getTargetVetexId,
    MyMessage(Current_Vertex_Id, TO_SEND)
         SET.clear
         TO_SEND.clear
    VOTE TO HALT
```

6. OPTIMIZATION TECHNIQUES

Giraph is designed to run the whole computation in-memory. Unfortunately, for large graph processing involving large message exchange computation result does not fit in-memory. Keeping the above drawback in mind, we applied below optimizations to our algorithm design ensuring only required set of messages flow across network during each superstep.

1. In multi-source breadth first search, for every message received by a node, parent id is updated with current vertex and forwarded to every connected node. However, if the source id received in current message is equal to the current vertex id, it is implicit that the message has traversed all graph nodes and hence it is not required to forward to connecting node. Including this condition at each node, we were able to reduce huge number of wasteful messages sent across nodes.

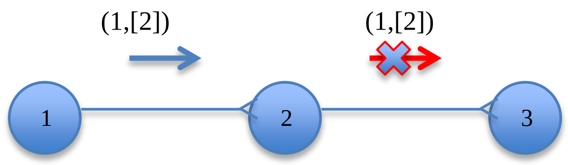


Figure 6 a. Illustrates first Optimization technique

2. Second condition was to restrict the child node from sending back the message it received from the parent node. Avoiding sending this message significantly decreased the number of messages sent across the graph.

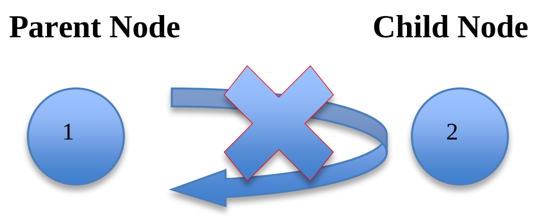


Figure 6 b. Message not sent back to the parent node

3. Initial implementation of the algorithm meant sending thousands of messages between the nodes, which used exceeded the maximum number of messages that a node could handle. To overcome this situation we combined messages in every iteration till we had array of messages at every node.

7. RESULTS

To validate the results of our algorithm, we tried to visualize the network graph we used to test the algorithm using NodeXL. We set certain Graph metrics on the Graph created to get the list of nodes with maximum edge betweeness.

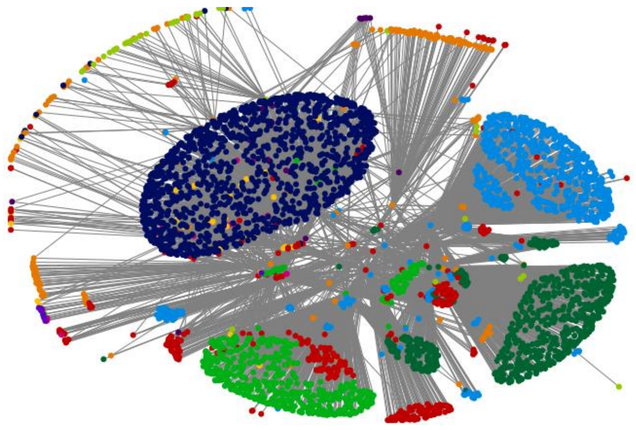
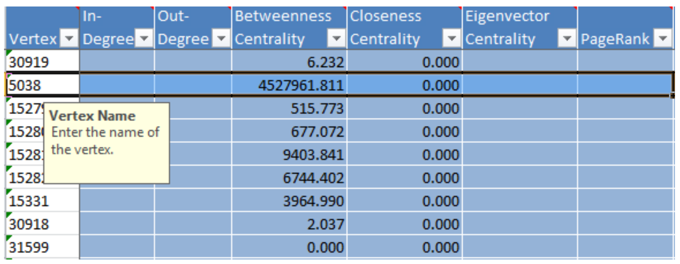


Figure7a. Cluster representation of Graph in NodeXL



Vertex	In-Degree	Out-Degree	Betweenness Centrality	Closeness Centrality	Eigenvector Centrality	PageRank
30919			6.232	0.000		
5038			4527961.811	0.000		
1527			515.773	0.000		
1528			677.072	0.000		
1528			9403.841	0.000		
1528			6744.402	0.000		
15331			3964.990	0.000		
30918			2.037	0.000		
31599			0.000	0.000		

Figure 7b. Node with Highest betweeness Centrality

We found that the results matched our Giraph results.

8. APPLICATIONS

Real world applications of the research work could be suggestions that one gets in a network like LinkedIn where in one can receive updates about the recruiters in a network that a most likely to connect a professional to an organization. This could also be used in email sending applications, to prompt that one important person who you might not want to miss out if you want to broadcast the information. It will help in understanding the information diffusion over a huge network.

9. LIMITATIONS

1. In multi-source breath first search, parent array of each node is created in form of an array structure stored at each node. The size of the array increases with increase in graph size. As Apache Giraph processing is in-memory, the size of the array puts a limit on the size of graph been processed by our algorithm. For larger graph, the array size increases which in turns demands for higher heap memory requirements.

2. The halt condition for our current algorithm is diameter of graph. It is required to pass the maximum number of supersteps required to process the graph. Ideally, the algorithm should exhaust automatically once all processing has been done.

10. ACKNOWLEDGMENTS

We would like to thank Prof. Jimmy Lin (Prof. and Associate Dean for Research at iSchool, Univ. of Maryland, College Park) for encouraging us to research in this space and constantly mentoring us throughout the project. We are also thankful to for his useful guidance. We also want to extend a vote of thanks to Jiaul Paik (Postdoctoral Research Associate Institute for Advanced Computer Studies (UMIACS) College of Computer, Mathematical, and Natural Sciences University of Maryland, College Park) to come up with this idea to work with

REFERENCES

[1] Despalatovic´, L., Vojkovic´, D., & Vojkovic´, T. (2014). Community structure in networks: Girvan-Newman algorithm improvement. Retrieved from http://docs.mipro-proceedings.com/cts/CTS_09_2540.pdf

[2] Girvan, M., & Newman, M. E. (2002). Community structure in social and biological networks. *Proceedings of The National Academy of Sciences*. doi:10.1073/pnas.122653799

[3] http://www.staff.science.uu.nl/~bisse101/Book/PSC/psc1_2.pdf

[4] Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., & Czajkowski, G. (2010). Pregel: a system for large-scale graph processing. doi:10.1145/1807167.1807184

[5] Giraph - Welcome To Apache Giraph! (n.d.). Retrieved from http://giraph.apache.org/

[6] Map reduce paper (Sanjay Ghemawat and Jeffrey Dean)

[7] Professor Lin's Paper (Design Patterns for Efficient Graph Algorithms in MapReduce)

[8] B. Panda, J. S. Herbach, S. Basu, and R. J. Bayardo. PLANET: Massively parallel learning of tree ensembles with MapReduce. In Proceedings of the 35th International Conference on Very Large Data Base (VLDB 2009), pages 1426–1437, Lyon, France, 2009.

[9] Professor Lin's Map Reduce Book

[10] http://web.mit.edu/6.976/www/handout/valiant2.pdf (BSP paper)

[11] Leskovec, J., Lang, K. J., Dasgupta, A., & Mahoney, M. W. (2008). Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters.Computing Research Repository.