IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received 9 May 2023, accepted 26 May 2023, date of publication 29 May 2023, date of current version 7 June 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3281266

RESEARCH ARTICLE

MapReduce for Graphs Processing: New Big Data Algorithm for 2-Edge Connected Components and Future Ideas

DEVENDRA DAHIPHALE, (Member, IEEE)
Department of Computer Science and Electrical Engineering, University of Maryland, Baltimore County, Baltimore, MD 21250, USA
e-mail: devendradahiphale@gmail.com

ABSTRACT Finding connectivity in graphs has numerous applications, such as social network analysis, data mining, intra-city or inter-cities connectivity, neural network, and many more. The deluge of graph applications makes graph connectivity problems extremely important and worthwhile to explore. Currently, there are many single-node algorithms for graph mining and analysis; however, those algorithms primarily apply to small graphs and are implemented on a single machine node. Finding 2-edge connected components (2-ECCs) in massive graphs (billions of edges and vertices) is impractical and time-consuming, even with the best-known single-node algorithms. Processing a big graph in a parallel and distributed fashion saves considerable time to finish processing. Moreover, it enables stream data processing by allowing quick results for vast and continuous nature data sets. This research proposes a distributed and parallel algorithm for finding 2-ECCs in big undirected graphs (subsequently called "*BiECCA*″) and presents its time complexity analysis. The proposed algorithm is implemented on a MapReduce framework and uses an existing algorithm to find connected components (CCs) in a graph as a sub-step. Finally, we suggest a few novel ideas and approaches as extensions to our work.

INDEX TERMS Graph, distributed, algorithm, connected component, big data, MapReduce, undirected, small star, large star, analysis, serial algorithms, parallel, cascaded jobs, Hadoop, streaming, connectivity.

I. INTRODUCTION

Graphs [1], [2], [3] model data into objects (also called nodes or vertices) and connections (also called edges) between these objects. This simple concept of representation model has tremendous applications in many fields. The field of graph theory is dedicated to finding efficient solutions for graph problems. There are many real-world applications that we map to graph problems, such as physical models (cities interconnected by roadways), social network models (users and connections between them that may represent friendship), neural networks (for both artificial intelligence and human brain studies), etc. graph theory has been well studied. There are several efficient algorithms for graph analysis for solving problems that can be represented as graphs. However, as graphs are becoming complex and massive due to advances in the big data domain and other data-related fields, processing large graphs efficiently is essential in real-time applications. This research primarily focuses on the problem of finding 2-ECCs [3] in big graphs in a parallel and distributed fashion. The proposed algorithm uses an existing graph convergence algorithm (also called a "Star Algorithm") [4] as a foundation that is used for finding connected components (considered 1-Edge by default) in a big graph.

Connectivity [4], [5] in a graph is one of the essential concepts in graph theory. The connectivity suggests the minimum number of vertices or edges which need to be removed from a graph to make that graph disconnected. Finding connectivity in a graph is a fundamental step in any graph analysis or mining problem. This work presents a new approach for finding 2-ECCs in undirected graphs using the MapReduce [6] framework. The term 2-edge connected implies at least two distinct paths (both ways) between any pair of vertices in a

The associate editor coordinating the review of this manuscript and approving it for publication was Alberto Cano.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

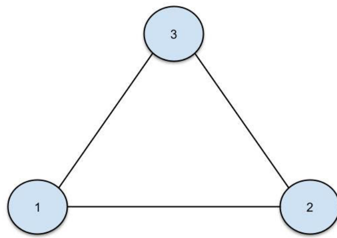

FIGURE 1. 2 edge connected graph.

given graph; for example, the graph presented in figure 1 is a 2-edge connected graph.

A. CONNECTIVITY IN GRAPHS

Graph connectivity [3] is a well-studied problem. It is one of the complex [5] problems of graph theory. Currently, there are many serial algorithms [7] available to find various types of connectivity relationships between vertices of a graph. The most commonly used serial algorithms for finding Strongly connected components [8], [9] (a type of connectivity relationship between vertices of a graph) in a graph are introduced by Tarjan's [10], Kosaraju's [11] and Vitter's [12]. Connectivity is a good measure of the resilience of a network. One way to measure connectivity is to find the minimum number of edges that need removal to make the graph disconnected. We leveraged this idea by dropping the edges step by step to find 2-ECCs in the presented algorithm.

For a graph to be K-edge Connected [3], there should be k distinct paths between any two vertices of that graph. For example, considering pair of vertices $< u, v >$, there are k different paths from u to v and the k distinct paths from v to u. This study focuses on 2-edge connectivity in big graphs and presents a parallel and distributed algorithm for finding 2-ECCs in huge undirected graphs.

B. IMPORTANCE OF FINDING 2-EDGE CONNECTED COMPONENTS IN PARALLEL

In graph theory, a graph G is said to be K-edge Connected graph when it remains connected after less than k edges are removed from any path between any pair of vertices of the graph G.

Finding 2-ECCs in small graphs is possible using the best-known algorithms [13], which run serially on a given input graph. However, using these serial algorithms to find 2-edge connected components for big graphs with millions of edges and vertices is inefficient. This is mainly because processing a graph with billions of edges or vertices becomes practical or practically useful. Furthermore, analyzing or mining big graphs, for example, Facebook Graph, Twitter, Google Graph, Linkedin Graph, etc., using the best-known serial algorithms will take considerable time to complete the processing. Therefore, the result cannot be used in real-time applications such as Weather Predictions, Prediction of Stock Market Prices, Social Media feeds, etc. Moreover, real-time targeted advertising would be failed with the batch processing of data.

Processing big graphs efficiently and in real-time is a fundamental requirement for big data processing applications. Finding connected components in a graph is an essential step for clustering [4] related entities represented as nodes or vertices in a graph. Processing big graphs in parallel and distributed fashion reduces graph analysis time by a great extent, which makes graph analysis applicable to solve more problems, and thus it enables many more applications.

C. EXISTING ALGORITHMS AND MOTIVATION FOR IMPROVEMENTS

Processing big graphs in parallel and distributed fashion helps make the processing faster. It also enables many streaming data applications, for example, analysis of tweet data in real-time, processing stock market data (from stock exchanges and social media), live events or sports data analysis, weather forecasting, etc. However, there is a scope for efficiency improvement in extracting critical information from graphs. Finding connected components is one of the essential features which helps in identifying Clusters or Stars [4], which is one of the relationships between vertices of a graph. Our foundational paper [4] gives a distributed algorithm to find connected components (1-edge connected components; however, 1-edge is never stated explicitly, it is considered by default) in big-size graphs and explains how it could be implemented using Hadoop (one of the implementations of the MapReduce framework). This paper presents an algorithm to find 2-ECCs in big undirected graphs. Finding 2-ECCs has many vital applications; for example, in a distributed network, paths connecting two different sub-networks are critical paths, and failure of such critical paths from the network can disconnect many users from accessing some crucial resources over the Internet. Therefore, avoiding such critical paths for network reliability is always better. Similarly, finding connectivity in the brain neurons would tell doctors a safe location on the head for incision in cases of brain surgeries. In such scenarios, we would require 2-edge connectivity or, in general, K-edge connectivity between the networks (whether it is a critical path in an Internet network or brain neurons); therefore, finding 2-ECCs becomes crucial.

D. OUR CONTRIBUTION

This section presents our contributions to finding 2-edge connected components (2-ECCs) in huge graphs. We propose a novel distributed and parallel algorithm for finding 2-ECCs in huge graphs. We also present a design and architecture of our proposal. We implemented our new algorithm on top of the Hadoop MapReduce framework. We designed and implemented five different MapReduce jobs and used them in a cascaded fashion. We did a detailed analysis of the proposed algorithm, which is mainly done in terms of time complexity. Finally, we presented the results and evaluations for various sizes of graphs. The results are presented in terms of the number of vertices and edges vs. the time taken for finding 2-ECCs. We also suggested some novel ideas as extensions to this work in the section Future Ideas. The readers are welcome to explore those ideas further.

As a prerequisite to testing our algorithm, we needed to design and implement a robust solution for creating big graphs. These big graphs are created with each vertex's degree equal or differ at most by 1. The degree of each vertex is kept approximately the same to avoid skewing the time required for processing a graph. Here are the specific contributions of this manuscript:

1) We propose a novel distributed and parallel algorithm for finding 2-ECCs in huge graphs. Furthermore, we illustrated the algorithm using an example and using diagrams.
2) We present a design and architecture of our proposal.
3) We implement our new algorithm on top of the Hadoop MapReduce [14], [15] framework.
4) We do a detailed analysis of the proposed algorithm, which is mainly done in terms of time complexity.
5) We present the results and evaluations for various sizes of graphs.
6) We suggest some novel ideas as extensions to this work in the section Future Ideas. The readers are welcome to explore those ideas further.
7) We design and implement a robust solution for creating big graphs.

This manuscript proposes a new distributed and parallel processing approach for finding 2-ECCs using one of the MapReduce [14] implementations. This new algorithm is built on top of an existing algorithm (presented in the paper [4]), which computes connected components in graphs. The fundamental idea is to calculate 2-ECCs by finding all the bridges [16] (Bridge is explained in the Background Survey section of this paper) in a graph and dropping those bridges one by one in each iteration until there are no more bridges left in the original graph. This is explained in great detail in the algorithm section of this manuscript.

Furthermore, we prove that our approach for finding 2-ECCs takes $O(E * log^2 V/P)$ MapReduce iterations (where graph contains $|E|$ number of edges, $|V|$ is the number of vertices and P is the number of edges checked for the bridge property in parallel). Assuming we have enough computing resources in a cluster to check all edges in parallel for the bridge property (checking if an edge is a bridge or not), the time complexity for our proposed algorithm comes down to $O(log^2 V)$ as P will be equal to E in the earlier equation. This manuscript is an enhanced version of my thesis from ProQuest [17]. The primary enhancements include 1. The algorithm made efficient, 2. Algorithm details are considerably improved 3. Explanation with better examples, 4. Updated the results and evaluation, and 5. Proposal of future ideas as extensions to the existing work. We see many benefits of incorporating ideas of distributed dynamic programming [18] in our solution.

E. OUTLINE OF THE MANUSCRIPT

Hereafter, the manuscript is organized as follows: Section II of this paper discusses the proposed idea's background survey and related topics. Section III. presents the proposed algorithm and its illustration for finding 2-ECCs in a given graph or forest. Section IV. is devoted to our design, architecture, and algorithm implementation. The time complexity analysis is done in section V. Section VI presents our results and their evaluations. The critical challenges that we faced while designing and implementing the idea and how those challenges are tackled are explained in section VII. Eventually, we stated a future scope and conclusion in sections VII. and VIII. respectively.

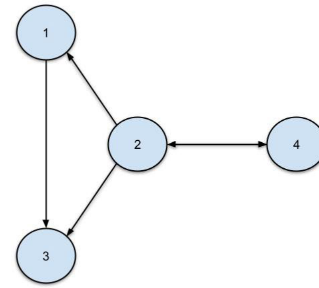

FIGURE 2. An example: directed graph.

II. BACKGROUND SURVEY

A. GRAPH

A graph $G = (V, E)$ is a structure that contains a set of nodes called vertices (V) and lines or arcs called edges (E). Vertices represent objects. Edges represent the connection between these objects. Graph G has $|E|$ number of edges and $|V|$ number of vertices, for example, the figure 3 shows a graph with five vertices: $\{1, 2, 3, 4, 5\}$ and the five edges are $(1, 3)$, $(2, 4)$, $(3, 4)$, $(3, 5)$ and $(4, 5)$.

1) UNDIRECTED GRAPH

An undirected [19], [20] graph is a graph where all the edges are bidirectional [21]. An undirected graph is sometimes called an undirected network; for example, figure 3 shows an undirected graph.

2) DIRECTED GRAPH OR DIGRAPH

In graph theory, a directed graph (or digraph) [22] is a graph that is made up of a set of vertices connected by directed edges (the edges point in a direction), often called arcs, for example, the figure 2 shows a directed graph.

B. DEGREE OF A VERTEX

The number of incident edges on the vertex v_i where $1 <= i <= |V|$ is called **degree** of the vertex v_i. For example, in the figure 3, the degree for the vertices with IDs $\{1, 2\}$ is 1; the degree for the vertices with IDs $\{3, 4\}$ is 3; and the degree for the vertex with ID $\{5\}$ is 2.

C. VERTICES AND EDGES

Vertices, also called nodes, in a graph can be interpreted as objects; vertices are usually represented with circles, and each

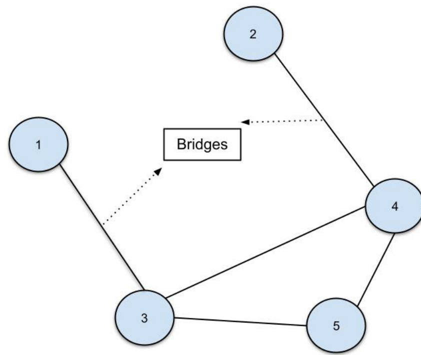

FIGURE 3. Graph with bridges.

vertex has a unique label. While preparing the input graph for feeding it to the implemented algorithm in the research work, all the vertices are labeled randomly to ensure that all the labels are unique. Edges are connections between vertices which are represented with lines or arcs. An undirected graph contains set of vertices $V = v_1, v_2, \ldots, v_n$ where $n = |V|$ and edges $|E|$, an undirected edge between v_i and v_j is represented as (v_i, v_j) where $1 <= i, j <= |V|$. For example, in the figure 3, the vertices are $\{1, 2, 3, 4, 5\}$ and the edges are (1, 3), (2, 4), (3, 4), (3, 5) and (4, 5).

D. CONNECTED COMPONENTS IN GRAPHS

A Connected Component [4] is a subgraph $G'(V', E')$ of graph $G(V, E)$, where $V' \in V$ and $E' \in E$ and G' each pair of vertices should be connected at least by one path.

Finding connected components in graphs is fundamental to clustering entities or finding new patterns among entities. Finding connected components using either breadth-first search (BFS) or depth-first search (DFS) can be done in linear time. By starting DFS or BFS search at each unvisited vertex v_i where $1 <= i <= |V|$, we can find all directly or indirectly connected vertices to v_i, all such vertices connected to v_i belong to the same Connected Component of a graph.

Finding connected components in parallel and distributed fashion using MapReduce can be done by alternatively and repeatedly applying the large star and small star operations; please refer to the primary reference for this work [4] for a detailed understanding of these two operations and how they are used to find connected components in a graph. We used these two operations as sub-steps to calculate 2-ECCs in a graph. This paper also explains the large and small star operations in section IV.

E. 2-EDGE CONNECTED COMPONENTS IN GRAPHS

A connected graph is called 2-edge connected if it does not have a bridge. A bridge (or cut arc) is an edge of a graph whose deletion increases the number of connected components, i.e., an edge whose removal disconnects the graph.

For example, figure 3 has two bridges 1. $< 1, 3 >$ and 2. $< 2, 4 >$. Because the graph has at least a bridge, it cannot be called a 2-edge connected graph. However, figure 1 has no bridges in it. Therefore, it is called a 2-edge connected Graph. In other words, at least two distinct paths between the nodes of any pair of vertices in the graph from figure 1.

F. BRIDGES IN A GRAPH

A bridge in a graph $G(V, E)$ is an edge $e' \in E$ when removed from the graph, $G(V, E - e')$ divides the graph into two different connected components. A bridge is also said to be a cut edge. A graph can be K-edge Connected where $k > 1$.

G. MapReduce

MapReduce [14] is a programming model or a theoretical approach and an associated implementation developed by Google for processing big data sets in a distributed and parallel fashion. Mapreduce operates in two phases: a Map Phase, in which input data is split into multiple chunks/splits, and for processing each chunk/split, a mapper (a programming thread) is spawned, which applies a user-defined function for processing, called the Map Function; and a Reduce Phase, which aggregates the data produced in the Map Phase to generate the final output. The aggregator threads apply a user-defined Reduce Function on the intermediate data; these workers/threads are called Reducers.

There are many MapReduce implementations; however, the most common and widely used is Hadoop. Hadoop Online Prototype [23], [24] is a modification to the traditional Hadoop. In conventional Hadoop, Map and Reduce Phases run sequentially - first Map phase finishes, and then Reduce phase starts. However, Hadoop Online Prototype, Map, and Reduce phases run simultaneously. As Hadoop Online Prototype is well-proven, the most popular, and efficient in processing big data sets, we decided to use it for the proposed algorithm for finding 2-ECCs in big graphs.

For example, the most famous word count example elegantly illustrates the computing phases of MapReduce. The task is to count occurrences of each word from an extensive document. To make the example easy to understand, consider the document to be processed contains the text as "*Fear leads to anger*; *anger leads to hatred*; *hatred leads to conflict*; *conflict leads to suffering*."'. The solution for counting the words in the document using MapReduce goes as follows (please refer the figure 4):

1) The document is divided into several chunks. The ID of a split is the input key, and the actual split (or a pointer to it) is the value corresponding to that key. Thus, the document is divided into a number of chunks, each containing a set of (*key*, *value*) pairs.
2) In the Map phase, the task scheduler (or the Primary server) starts M mappers where each mapper is responsible for processing one of the chunks from the input data. The mappers will scan the document and emit records (key and value pairs) for further processing. According to the user-defined Map function, the Mappers will only map each word to value one here. For example, this mapper will emit the following records

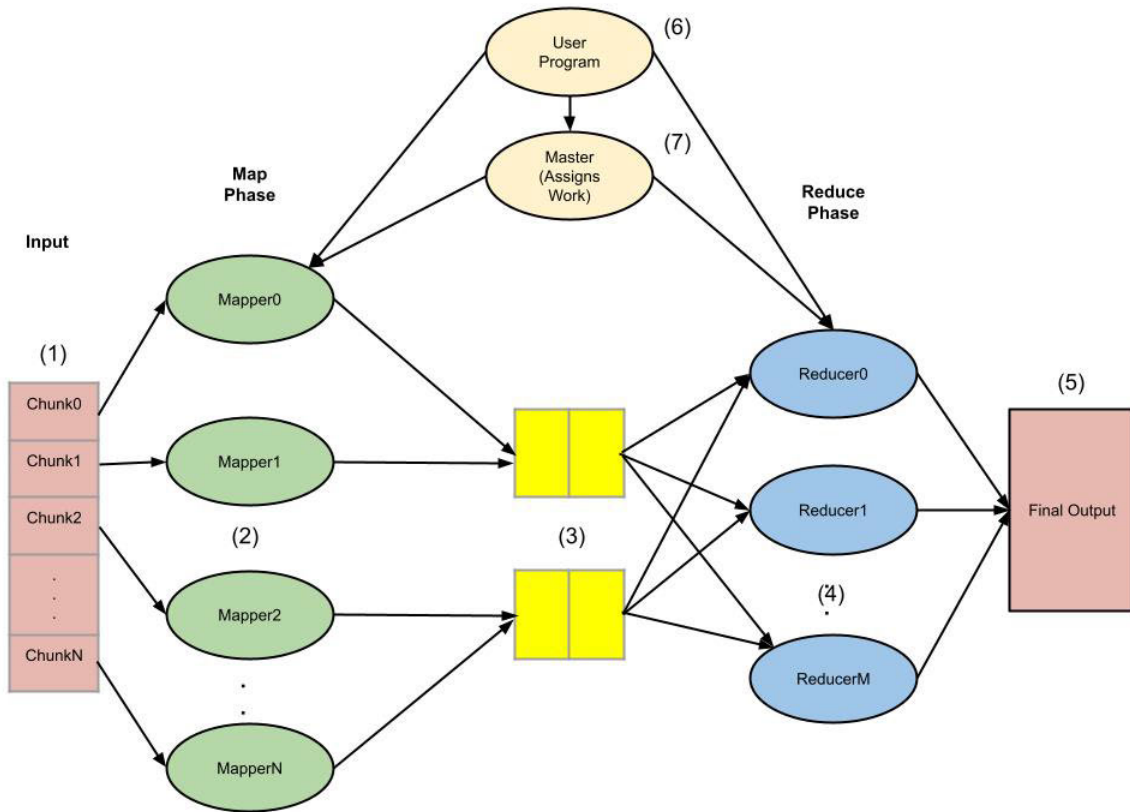

FIGURE 4. Architecture: MapReduce framework.

TABLE 1. Word count example: first mapper thread/worker's output.

$WordString$	$Frequency(Mapped\ to\ 1)$
Fear	1
leads	1
to	1
anger	1

TABLE 2. Word count example: final output - all words with their frequency.

$WordString$	$Frequency/Count$
anger	2
conflict	2
Fear	1
hatred	2
leads	4
suffering	1
to	4

when a Mapper has a chunk with a part of the document as "*Fear leads to anger*".

3) The set of intermediate (*key*, *value*) pairs is then "pulled" by the Reducers that the Mappers produced. These intermediate records are kept in the local storage where the mappers run.
4) The Map phase is now finished, and the Reduce phase has started. The Reducers will now aggregate all the records with the same keys. The final output will be as follows:

   This suggests that the word "anger" has occurred two times in the input data, and the word "leads" has occurred four times in the input data, and so on.
5) This item in the figure 4 shows the final output of the MapReduce job. The final output is also stored in HDFS.
6) This item in the figure 4 is the application program that launched/started the word count MapReduce job.
7) This item is the primary server or the task tracker. This is responsible for starting different workers from the job and tracking their progress.

H. HADOOP

Hadoop [15] implements the MapReduce programming model developed by Apache. The Hadoop framework is used for batch and stream processing big data sets on a physical cluster of machines. It incorporates a distributed file system called Hadoop Distributed File System (HDFS), a common set of commands, a scheduler, and the MapReduce evaluation framework. Hadoop is famous for processing massive data sets (also called big data), especially in social networking, targeted advertisements, internet log processing, etc.

I. SPARK

Spark [25] MapReduce is a unified analytics engine for large-scale data processing. It provides high-level APIs in

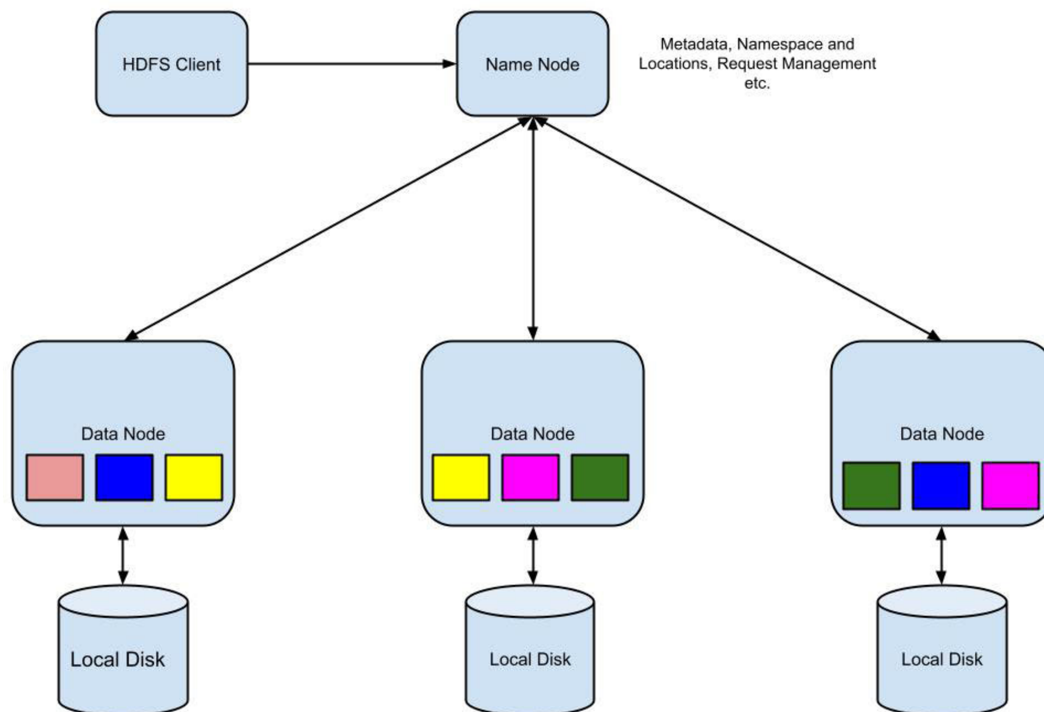

FIGURE 5. Architecture: Hadoop distributed file system (HDFS).

Java, Scala, Python, and R and an optimized engine that supports general execution graphs. It also supports a rich set of higher-level tools, including Spark SQL for SQL and structured data processing, MLlib for machine learning, GraphX for graph processing, and Spark Streaming. Spark MapReduce is a powerful tool for processing large-scale data sets. It is fast, scalable, and easy to use. It is a good choice for various data processing applications, including batch processing, streaming processing, machine learning, and graph processing.

J. 2-EDGE CONNECTIVITY USING SERIAL ALGORITHMS

The 2-edge connectivity [26], [27] of an undirected graph can be determined in linear time [27]. Many sub-problems of the 2-Edge connectivity, like finding strong bridges and articulation points, can be calculated in linear time based on the dominator tree algorithm proposed in reference [28]. Determining the 2-Edge connectivity of a directed graph efficiently is challenging. A naive algorithm for finding the 2-Edge connectivity can be visualized as removing strong bridges from a graph one by one, repeating this procedure until no bridges are left in the graph. Strong bridges from a graph can be calculated in linear time, and this process continues $O(N)$ times; thus, the running time of this algorithm is $O(N * M)$ where $|N|$ is the number of vertices and $|M|$ is number edges in a graph. Jaberi [29] proposed an algorithm that maps the problem of determining the 2-vertex connected component of directed graphs to determine 2-Vertex connected components in undirected graphs (they used a technique of reducing a new problem to an existing problem first and then solve it). Still, they did not present the running time analysis. Later Erusalimskii and Svetlov [30] showed the algorithm with $O(N*M^2)$ time complexity, which is asymptotically more than the naive algorithm. Georgiadis et al. presents a dominator tree-based algorithm with a running time of $O(M * N)$. 2-edge connectivity is a sub-problem of the bigger problem of finding the K-edge connectivity or k-vertex connectivity [3]. Matula [31] gave an algorithm to determine the edge-connectivity in $O(mn)$ time. He also showed that given k in advance, testing whether a graph is k-edge-connected could be done in $O(K*N^2)$ time. Nagamochi and Watanabe [32] gave an $O(N * min(K, N, \sqrt{M}) * M)$ time algorithm for finding all k-edge connected components in a direct or undirected graph given k in advance.

K. HADOOP DISTRIBUTED FILE SYSTEM (HDFS)

HDFS [15] is a distributed file system that provides reliable and scalable data storage. It is designed explicitly for spanning large clusters on the commodity hardware. Hadoop uses a hadoop distributed file system for data storage. The input data is pulled from the HDFS for processing, and results are kept in HDFS by default. In addition, the HDFS also acts as the intermediate staging area for the intermediate results of multiple MapReduce jobs and in various phases of the same MapReduce job.

The figure 5. shows a fundamental architecture of HDFS. HDFS has a Primary/Secondary architecture. An HDFS cluster consists of a single NameNode. The NameNode is a Primary server that manages the file system namespace and regulates access to files by clients. The DataNodes manage storage attached to the nodes that they run on. HDFS exposes

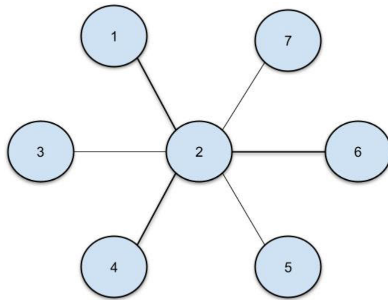

FIGURE 6. A star or cluster.

a file system namespace and allows user data to be stored in files. The DataNodes are responsible for serving read and write requests from the file system's clients, and also they perform block creation, deletion, and replication upon instruction from the NameNode.

III. OUR ALGORITHM PROPOSAL AND ILLUSTRATION

An algorithm to find connected components in big graphs in parallel [4] is extended to find 2-ECCs [3]. The fundamental idea behind the proposed approach is that a 2-edge connected component is also a connected component without bridges. The proposed algorithm is implemented on top of the Hadoop [6], [23] framework. First, all the bridges are removed, one at a time, from a graph to be processed. By eliminating all bridges, one by one, finding the 2-ECCs problem is reduced to the problem of finding connected components. Then, a distributed and parallel algorithm to find connected components is repeated until the input graph converges (no more bridges are left in the graph).

A. CONVERTING GRAPH INTO STARS OR CLUSTERS

Let $G = (V, E)$ be an undirected graph with $|V|$ number of vertices and $|E|$ number of edges where each vertex $v \in V$ is labeled randomly to any integer number between 1 and $|V|$ inclusive and denoted as l_i; and each edge $e \in E$.

1) SIMPLEST VERSION OF THE STEPS OF THE DISTRIBUTED ALGORITHMS

Note that the algorithms, in terms of Map and Reduce functions, are presented later in the current section. The following are the short steps to get a quick glimpse of the proposed algorithm.

1) Take the graph G to find 2-ECCs.
2) Select an edge e from the Graph G to check if e is a bridge.
3) Remove the edge e from the graph G to get the graph G'.
4) Find the number of connected components (C') in the new graph G'.
5) Compare the number of connected components (C) in the graphs G with that (C') in G'. If $C' > C$ then the edge e is a bridge in the graph G.
6) If the edge e is not a bridge, add the edge e back to the G' (i.e., henceforth process the graph G instead of the generated graph G').
7) Stop if the graph G and the graph G' are the same and all the bridges from the graph are eliminated.
8) If the edge e is a bridge, then do $G = G'$.
9) Go to step 2 if the graph is not converged yet.
10) Get the final Stars or Clusters from the latest version of the Graph. Those are the 2-ECCs.

Figure 6 shows a Star or Cluster where all the nodes are connected to a single node (the star's center). For converting the graph G into a Star or Stars, where each star represents a connected component, the large star and small star operations are applied on graph nodes. These two operations are referred from the foundational paper [4]. The large star and small star operations are explained in our manuscript in section IV-C. The large and small star operations are alternatively and repeatedly applied until the input graph converges (i.e., the graph no longer changes after using any large Star and small star operations).

B. FINDING BRIDGES IN A GRAPH

In a graph $G(V, E)$, to verify whether an edge e, where $e \in E$, is a bridge, stars are counted in that graph with and without the edge e. If the number of stars or clusters (computed using the hadoop framework in a distributed fashion) counted after removing the edge e from the graph G is one more than the stars counted before removing the edge e from graph G implies that the edge e is a bridge in the input graph. Therefore, the bridges from the original graphs are eliminated, one at a time.

The Map and Reduce functions for removing an edge from the graph are as follows (presented in Algorithm 1. below). The removed edge will be under examination for the bridge property. In an edge list, u and v are the endpoints of an edge where $u > v$. In the map function, all edges are reversed, which will be used to find the degree of the vertices in the Reduce Phase. $\Gamma(u)$ represents all neighbours of vertex u.

1) ALGORITHM 1: AN EDGE REMOVER FROM AN INPUT GRAPH

01: **Input** : Edge list
02: **Input** : *EDGE_COUNTER* in the context of the MR job
03: **Input** : *TARGET_EDGE_NUMBER* in the context of the MR job
04: **Map** $\langle u; v\rangle$:
05: Emit $\langle v; u\rangle$.
06: **Reduce** $\langle u; \Gamma(u)\rangle$:
07: **For** $\forall$ m $\in \Gamma(u)$
08: **If** EDGE_COUNTER != TARGET_EDGE_NUMBER
09: Emit $\langle v; m\rangle$ for all v where $l_v > l_u$.
10: increment(EDGE_COUNTER)
11: **Else**
12: **If** degree(m) = 1
13: Emit(m, -1)
14: **If** degree(u) = 1
15: Emit(u, -1)

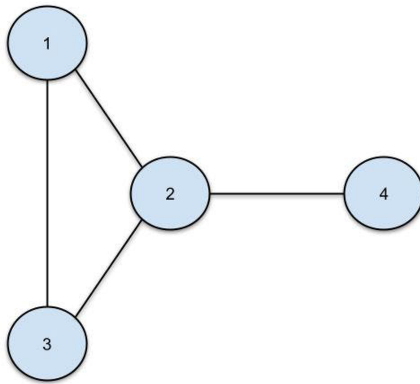

FIGURE 7. Input graph.

NodeID	Neighbours
1	2, 3
2	1, 3, 4
3	1, 2
4	2

FIGURE 8. Adjacency list.

C. COMPUTING CONNECTED COMPONENTS IN GRAPHS

Each star/cluster that we get after converting the input graph by applying the large star and small star operations represents a connected component. The number of stars denotes the number of connected components in the graph. Moreover, all vertices from a star belong to the same connected component.

D. COMPUTING 2-EDGE CONNECTED COMPONENTS IN A GRAPH

After checking if an edge is a bridge or not; and removing all the bridges from the edge list for the input graph (the original graph), the small star and sarge star operations are performed alternatively and repeatedly until the input graph converges [4]. The converge means any of the Large or the Small star operations on the results graph become ''No Operation'' - Noop (no edges can be dropped anymore to preserve the 1-edge connectivity of the graph - to keep the graph connected). The stars are the output of the last step representing all the 2-ECCs in the graph, where one star/cluster represents one 2-edge connected component. The format of input given to the *BiECCA* algorithm is a file organized as an Adjacency List [33] in which the first column of each line contains NodeId/VertexId, and all neighbors will be on the same line after NodeId followed by a Tab space. Furthermore, all the neighboring nodes are separated by a comma for parsing purposes.

The output of the *BiECCA* algorithm is clusters of nodes where one cluster corresponds to one 2-ECCs. The output file is created at user defined folder location (on HDFS),

Star #	Nodes in Star
1	$\{1, 2, 3\}$
2	$\{4\}$

FIGURE 9. Final output for the given input.

which contains the following format (assuming the input graph contains N number of 2-ECCs).

$$\begin{gathered}\langle Cluster1\rangle\\ \langle Cluster2\rangle\\ \langle Cluster3\rangle\\ \dots\\ \dots\\ \dots\\ \langle ClusterN\rangle\end{gathered}$$

The final output for the input graph from figure 7 contains two clusters/stars as shown below:

The output from the figure 9 consists of 2 different stars or clusters, nothing but the two 2-ECCs. The vertices $\{1, 2, 3\}$ belong to the first 2-ECC and a vertex $\{4\}$ belongs to the second 2-ECC. The output implies two distinct paths between any pair of nodes in the sub-graph with vertices $\{1, 2, 3\}$. However, let's look back at the original input graph. There are no two distinct paths between any node from the first cluster to vertex ID 4 or vertex ID 4 to any node from cluster 1, implying that the overall graph is not 2-edge connected, but the two output stars individually are.

E. ALGORITHM TO FIND 2-ECCs

1) ALGORITHM 2: USING TWO-PHASE ALGORITHM

01: **Input** : Graph G(V, E) as an adjacency list
02: Convert Adjacency List to Edge List
03: **For** each edge e from graph G
04: Apply Algorithm 1 (an edge remover) on G to get G'
05: $G'(V', E') = G(V, E - \{e\})$
06: **Repeat**
07: **Repeat**
08: Large-star
09: **until** Converges
10: Small-star
11: **until** G Convergence
12: C = Connected Components (Stars) in G.
13: C' = Connected Components (Stars) in G'
14: **If** $C' > C$
15: The e is a bridge.
16: $G = G'$
17: **End if**
18: **End for**
19: Convert edge list of G to clusters or stars.
Note : Lines 06 *to* 11 are referred from [1].

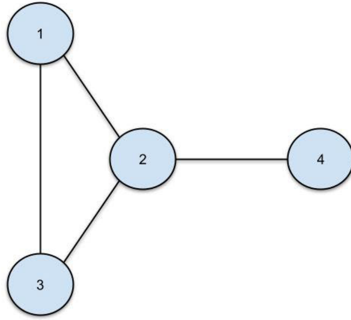

FIGURE 10. Input graph for illustration.

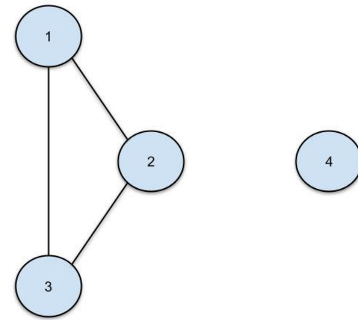

FIGURE 11. Dropping an edge $\langle 4,\ 2\rangle$ - iteration #1.

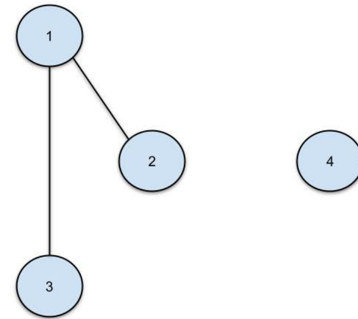

FIGURE 12. Applying large star operation on every node - iteration #1.

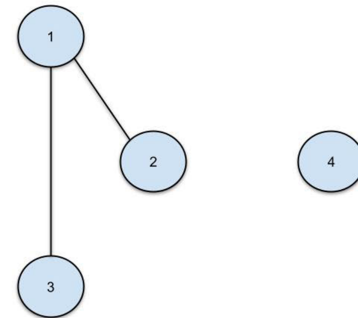

FIGURE 13. Applying small star operation on every node - iteration #1.

2) ALGORITHM 3: USING AN ALTERNATE ALGORITHM

01: **Input** : Graph G(V, E) as an adjacency list
02: Convert Adjacency List to Edge List
03: **For** each edge e from graph G
04: Apply Algorithm 1 (an edge remover) on G to get G'
05: $G'(V', E') = G(V, E - \{e\})$
06: **Repeat**
07: Large-star
08: Small-star
09: **until** G Converges
10: C = Connected Components (Stars) in G.
11: C' = Connected Components (Stars) in G'
12: **If** $C' > C$
13: The e is a bridge.
14: $G = G'$
15: **End if**
16: **End for**
17: Convert edge list of G to clusters or stars
Note : Lines 06 *to* 09 are referred from [1]

F. ALGORITHM ILLUSTRATION WITH AN EXAMPLE

The following example shows one iteration of the outer *for* loop from the algorithm presented above for dropping one edge between vertex with label 2 and vertex with label 4, i.e., to verify whether the dropped edge $\langle 2,\ 4\rangle$ is a bridge.

Step 1: The graph shown in figure 10 is given as the input. Figure 8 shows the corresponding adjacency list.

Step 2: Next, the adjacency list (figure 8) is converted into the edge list as follows. Note that we picked emitting only $\langle v,\ u\rangle$ edges where node ID for v is greater than that of u to avoid redundancy.

$$\langle 2,\ 1\rangle$$
$$\langle 3,\ 1\rangle$$
$$\langle 3,\ 2\rangle$$
$$\langle 4,\ 2\rangle$$

Step 3: Next, compute the connected components for the given input graph. For example, suppose C is the number of connected components; therefore, $C = 1$ because there is only one connected component in the input graph.

Step 4: Select an edge randomly and remove it from the input graph to check if it is a bridge. For example, pick an edge $\langle 4,\ 2\rangle$ randomly and remove it. After removing the selected edge, the resulting graph is shown in figure 11.

Step 5: Apply the large and small star Operations (please refer figure 12 and 13) alternatively and repeatedly until the input graph does not change further, i.e., the graph converses. After this step, the final graph is shown in figure 12. Finally, please refer to an example of a more complex graph from reference [17], section Appendix-B.3, and step by step process about how the Large and Small operation reduces the input graph to the convergent point.

Note: The final graph will not change further by applying any of the large star or small star operations.

Step 6: Compute the number of connected components in the new converged graph. C' denotes the new commented components. Therefore, $C' = 2$. In the first iteration of the proposed algorithm, we can conclude that an edge $\langle 4,\ 2\rangle$ is a bridge because when it is removed, $C' = C+1$, and therefore, this Edge will be removed permanently from the graph for further iterations of the algorithm.

Finally, after finding and removing all the bridges from the original graph, the remaining connected components are nothing but the 2-ECCs in the original graph.

Output Output of this example will have two lines/stars/clusters, each corresponding to one 2-edge connected component. The *BiECCA* algorithm's output for the input graph (please refer to figure 7) is as follows:

{1, 2, 3}
{4}

IV. DESIGN, ARCHITECTURE AND ALGORITHM IMPLEMENTATION USING MapReduce

Figure 14 shows the various components (or MapReduce jobs) used for processing. This section explains our design and architecture, and illustrates all the components of the diagram.

A. INPUT GRAPH GENERATION

First, we developed a program (subsequently called "Random Graph Generator") for generating massive graphs. These graphs will be used as inputs by the proposed algorithm for testing. The "Random Graph Generator" is a program written in C++ language to generate graphs with random connectivity (however, the degree of each vertex is tried to keep the same using an equal probability function. The maximum difference between the degree of any two vertices is kept 1) for testing and evaluating algorithm extensively. The "Random Graph Generator" program takes a number of vertices and a number of edges as inputs and generates a graph as an output. Based on the user's input about a number of edges and a number of vertices, this program generates a random graph as an Adjacency List [33]. The generated Adjacency List is stored in the Local File System as a text file. This file is then up-streamed to the hadoop distributed file system for sourcing it as an input to the proposed *BiECCA* algorithm.

B. GRAPH STAGING AREAS IN MapReduce JOBS

Once input is uploaded from a Local File System to a hadoop distributed file system, all the intermediate and final graph files generated for the MapReduce Jobs are stored in a hadoop distributed file system. Therefore, the initial staging area for the graph is Local File System (before uploading it to the HDFS); however, input (just before starting an MR job), intermediate output, and final output staging areas for all the MapReduce jobs are HDFS.

C. ALL IMPLEMENTED MapReduce JOBS

1) ADJACENCY LIST TO EDGE LIST CONVERTER (MapReduce JOB-1)

There are many approaches to partition [34] graphs. Because the output of the MapReduce Job-1 is an edge list, it can be split into multiple chunks, where each chunk can have a set of edges. The MapReduce Job-1 only has the Map phase (no Reduce phase). Therefore, it has all task-managing workers/trackers and Mappers only. The MapReduce Job-1 takes an Adjacency List as an input graph to be processed (in a file format) and produces an Edge List where each row contains a $\langle key, value\rangle$ pair. This pair $\langle key, value\rangle$ represents the endpoints of an edge from the input graph. To deal with each edge only once, we emit one record out of $\langle u, v\rangle$, and $\langle v, u\rangle$. For further requirements in the jobs' pipeline and simplicity, only records $\langle key, value\rangle$ where $key > value$ are emitted, mainly to encounter an undirected or bi-directional edge only one time instead of two times.

Adjacency List to Edge List

01: **Map** $\langle u\rangle$:
02: For all neighbors of u:
03: Emit $\langle u; n\rangle$ n where $l_u > l_n$.

2) LARGE STAR OPERATOR (Mapreduce JOB-2)

The MapReduce Job-2 contains Mappers, Combiners, and Reducers, along with the other job managing and tracking workers. The MapReduce Job-2 takes the Edge List as the input generated by the MapReduce Job-1 and applies the large star [4] operations on it. Please refer to figure 15 for one large star [4] operation on NodeId 3 (this is a minimal step in the algorithm). For a detailed description of this operation and the below algorithm, please refer to the foundation paper [4], [17].

The Large Star Operation

01: **Map** $\langle u; v\rangle$:
02: Emit $\langle u; v\rangle$ and $\langle v; u\rangle$.
03: **Reduce** $\langle u; \Gamma(u)\rangle$:
04: Let $m = arg\ min_{v \in \Gamma^+(u)} l_v$.
05: Emit $\langle v; m\rangle$ for all v where $l_v > l_u$.

3) SMALL STAR OPERATOR (Mapreduce JOB-3)

The MapReduce Job-3 also contains Mappers, Combiners, and Reducers, along with the other job managing and tracking workers. The MapReduce Job-3 applies the small star operations on the output of MapReduce Job-2. Please refer to figure 16 [4] for one small star operation on one node (this is another minimal step in the algorithm). For a detailed description of this operation and the below algorithm, please refer to the foundational paper [4], [17].

The Small Star Operation

1: **Map** $\langle u; v\rangle$:
2: **if** $l_v \leq l_u$ **then**
3: Emit $\langle u; v\rangle$.
4: **else**
5: Emit $\langle v; u\rangle$.
6: **end if**
7: **Reduce** $\langle u; N \subseteq \Gamma(u)\rangle$:
8: Let $m = arg\ min_{v \in N \cup \{u\}} l_v$.
9: Emit $\langle v; m\rangle$ for all $v \in N$.

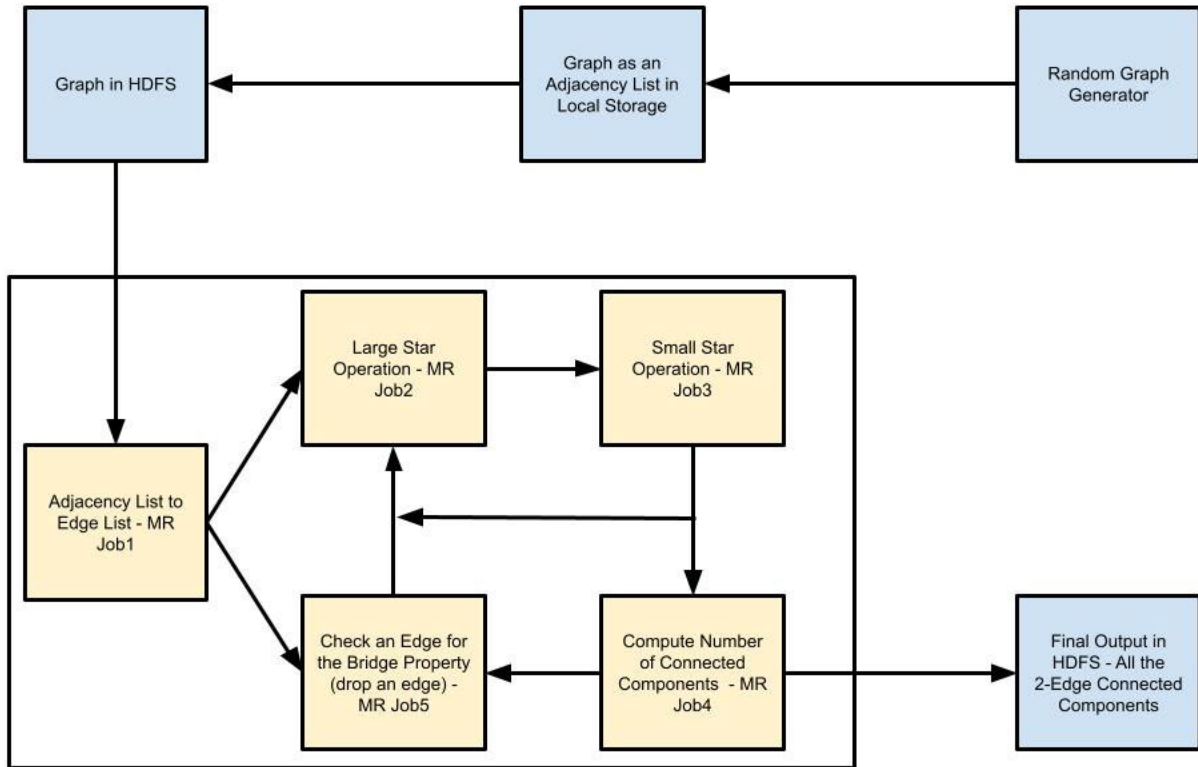

FIGURE 14. Architecture: component diagram.

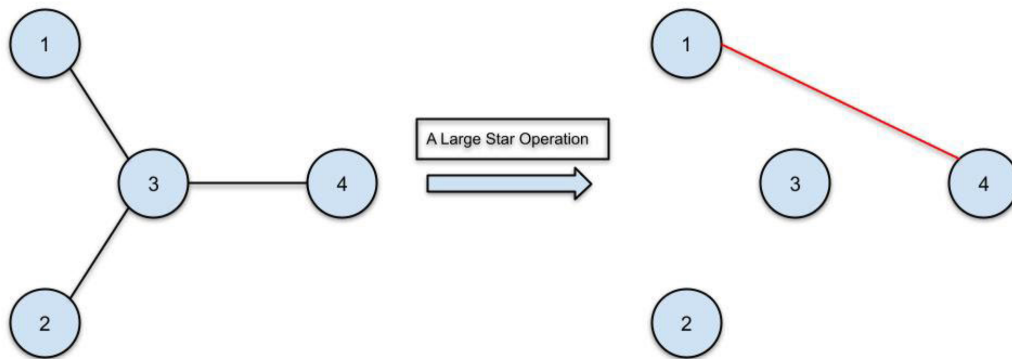

FIGURE 15. Large star operation on node 3.

4) THE GRAPH CONVERGENCE CHECKER(MapReduce JOB-4)

After alternatively and repeatedly applying the large and small star operations on the input graph, the graph eventually must converge. This MapReduce job converts the new graph to clusters of nodes for computing the number of stars/-clusters at the end of each iteration. When the input graph converges, the number of connected components calculated in the earlier step should equal the number of connected components calculated now. If the graph has converged, then the clusters (Stars) [4] are nothing but the 2-ECCs in the original graph. If the graph has not converged, the dropped edge is again added back to the previous graph (the graph before feeding it to the MapReduce Job-2) and sent to the MapReduce Job-5 for further processing.

5) ONE EDGE REMOVER - BRIDGE PROPERTY CHECKER (MapReduce JOB-5)

The MapReduce Job-5 takes the output of the MapReduce Job-4 and removes a specified (a random edge from the most recent version of the graph) edge from the given edge list. MapReduce Job-5 handles all cases when a specified edge is dropped from the graph and creates a correct new edge list as a new graph or a Forest. The new graph is called a Forest

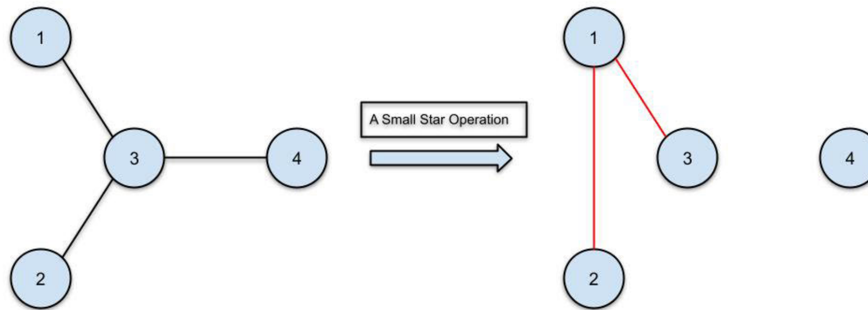

FIGURE 16. Small star operation on node 3.

because the removed edge might disconnect the graph and make some vertices disconnected. Whether the removed edge is a bridge is verified by checking if a number of connected components [4] are increased after removing the edge from an input graph. The basic idea is when a specified edge is dropped from a connected graph, if the number of connected components in the new graph is increased by 1, it implies that the dropped edge was indeed a bridge in the original graph.

6) INTERPRETING OUTPUT AS 2-EDGE CONNECTED COMPONENTS

After finding and removing all the bridges from the input graph (by alternatively and repeatedly applying the large star and small star [4] operations until no other bridges exist in the input graph, i.e., the input graph converges. Then, the remaining edge list is given to MapReduce Job-4. This MapReduce job is responsible for converting the final edge list into the clusters of connected components. The number of clusters or groups [4] formed represents the number of 2-ECCs where a group of vertices represents a 2-ECC.

V. ALGORITHM ANALYSIS: NUMBER OF MapReduce ITERATIONS

Theorem: The number of MapReduce iterations for finding 2-ECCs required is in order of $O(E*log^2V)/P$ where E is the number of edges, V is the number of vertices and P are the number of edges that are removed and processed in parallel for finding bridges in the given input graph. MapReduce iterations and rounds refer to the same term used interchangeably throughout the paper.

In Algorithm 2 presented before, a two-phase algorithm from [4] is executed for every edge in a graph. An edge is removed and repeated for each iteration of the outer loop until all the edges are done checking for the bridge property. Therefore, the outer loop runs E iterations, where E is the number of edges in the graph. We know from theorem 1 of [4] the number of MapReduce rounds to execute the two-phase algorithm is $O(log^2V)$.

Therefore, the number of MapReduce rounds for this algorithm is the number of edges times the number of MapReduce rounds for the two-phase algorithm. Therefore, the number of MapReduce rounds required for the algorithm is $O(E*log^2v/P)$ where P number of edges are processed in parallel for checking the bridge property.

VI. RESULTS AND EVALUATION

A. INPUT DATA AND SIZES

Input graphs are generated using a program that takes a number of edges and vertices as input parameters and generates graphs as an Adjacency List [33] in a file. This file is given as input for finding 2-ECCs. Testing is done on different sizes of graphs generated by the Random Graph Generator that was developed as a pre-requisite to this work. For example, the smallest graph used has four vertices and four edges; however, the most extensive graph tested has more than 1 million vertices and more than 2 million edges.

B. HADOOP FRAMEWORK

The proposed algorithm is well fit for large-size graphs. It is meant to be executed in a parallel and distributed fashion. Hadoop is one of the MapReduce implementations for processing big data sets distributedly. This is possible because the problem of finding 2-ECCs fits into the MapReduce model with multiple MapReduce jobs in the pipelined manner (cascaded MapReduce jobs [6]) for processing.

C. RESULTS FOR DIFFERENT SIZES OF RANDOM GRAPHS

The figures 17 and 19 show that when the inputs are small graphs, the time required to find 2-ECCs does not change notably. This observation is because the initialization/setup time for the MapReduce jobs is much more than the actual processing time. In other words, if the setup time is relatively more than the actual graph processing time and also processing time itself is significantly less, the overall difference in the performance for small graphs is not expected to be noticeable (this is because the setup time for all the job is going to be approximately the same). This is because the setup divides the graph into multiple chunks, initializes resources, spawning manager/tracker/processing threads, etc. Furthermore, the input graphs size should be large to get the initial setup time amortized [35]. This makes sense because the proposed approach is for processing big data sets on distributed processing frameworks. In line with these observations, when

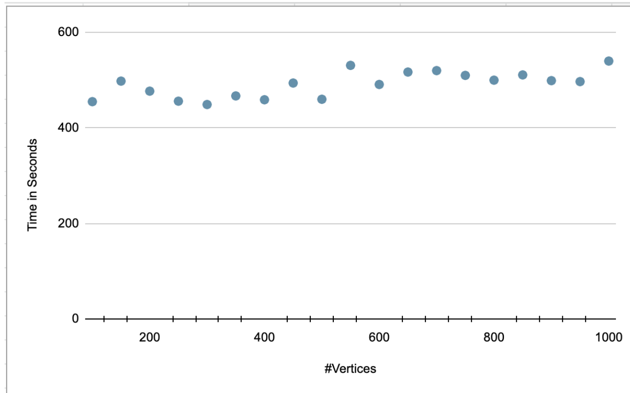

FIGURE 17. Number of vertices vs. time in seconds.

large-size graphs are used, the setup and initialization time gets amortized over a long period. Therefore, we can see noticeable differences in the processing time.

When large graphs are fed as inputs to the BiECCA algorithm implementation, we can observe that the time required to find 2-ECCs increases as the number of vertices and edges in the graph increases, keeping all other configurations the same, this is demonstrated by the figures 18 and 20. The configurations include the number of Mappers [14], the number of Reducers [14], the approximate degree of each vertex (structure of connectivity in the graph), and other computing resources such as CPU, Memory, Processor type.

Figure 21 compares the number of MapReduce iterations calculated theoretically using worst-case analysis and the number of MapReduce iterations calculated practically for different randomly generated graphs. The worst-case analysis for MapReduce considers the graph to be a single line where the end vertices of the line have degree one, and the middle vertices have degree two, where the depth of the graph will be equal to a number of vertices. However, in real-world scenarios, graph structure could be anything. Also, while generating random graphs, the degree of all vertices in a graph is kept constant using equal probability distribution while generating edges in a graph. The number of MapReduce iterations is a function of the depth of a graph where the depth of a graph is *max of all min distances of all pair of vertices in the graph*. The worst-case depth of a graph G and the real-world graph depth of G' can have a considerable difference even if G and G' have the same number of vertices and edges. Therefore the number of MapReduce rounds practically calculated and calculated theoretically differ considerably - in short, the algorithm performs much better in real-world use cases.

Tables 3 and 4 show the statistics for different graph sizes, given the graph as an input to the proposed *BiECCA* algorithm. Here, the number of records implies the size of the edge list to be processed. The time is given for one iteration of the algorithm; however, it will be increased by a factor of the number of edges to be checked for the bridge property. Assuming P edges can be checked in parallel, the time column in tables 3 and 4 will be multiplied by $m \div P$ where m is the number of edges in the input graph.

TABLE 3. Run time for different sizes of Input graphs with 1 Mapper and 1 Reducer.

$Index$	$No.\ of\ Inputs\ Records$	$Time/seconds$	$MR\ Rounds$
1	8	144	4
2	200	410	4
3	400	398	6
4	800	410	4
5	2000	400	5

TABLE 4. Run time for different sizes of input graphs with 5 Mappers and 5 reducers.

$Index$	$No.\ of\ Inputs\ Records$	$Time/seconds$	$MR\ Rounds$
1	40000	249	5
2	80000	271	5
3	120000	319	6
4	160000	339	6
5	200000	351	6
6	800000	389	7
7	4000000	869	9

The table 3 shows the time required for processing small-size graphs with one Mapper thread and one Reducer thread (we chose one Mapper and one Reducer because the graphs are small). As mentioned in figures 17 and 19 illustration, because the setup time of the MR jobs is more than the actual processing time (finding 2-ECCs), all jobs take approximately the same time to finish. However, table 4 shows the time for one iteration of the proposed algorithm of finding 2-ECCs increases with an increase in the graph size. This happens because workers' setup and initialization time from MR jobs is amortized over time.

D. VERIFICATION OF THE RESULTS, PRACTICALLY

The generated files are compared with the output generated for the same graph using well-known serial algorithms to practically check for the *BiECCA* algorithm's correctness. Although the serial algorithms take a considerable amount of time to solve the same problem, they are used for checking the correctness of the distributed algorithm only (though we could not run a massive graph with the serial algorithms because the wait time to finish jobs was in terms of days). Processing big graphs using a serial algorithm becomes impossible and difficult (beyond a certain number of vertices or edges).

VII. CHALLENGES AND SOLUTIONS

A. CHALLENGES

One of the algorithm's steps is to drop an edge from the graph and repeat the process for every edge. Therefore, removing edges one at a time was challenging for the following reasons.

1) Dropping an edge from an edge list that connects vertices with a degree more than one:
In this case, we can drop the key and the associated value (i.e., the edge between the node ID "key" and

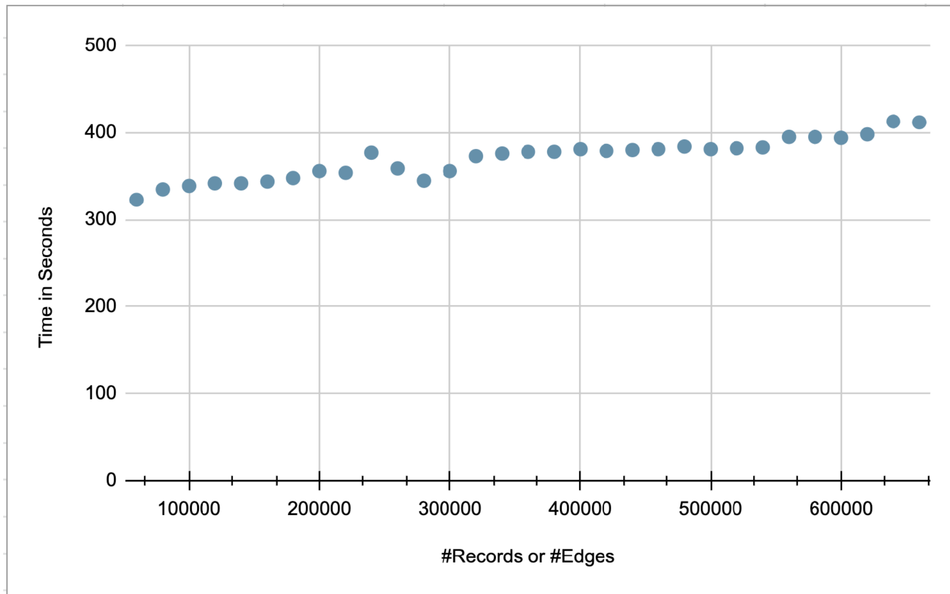

FIGURE 18. Input records/edges vs. time in seconds for large graphs.

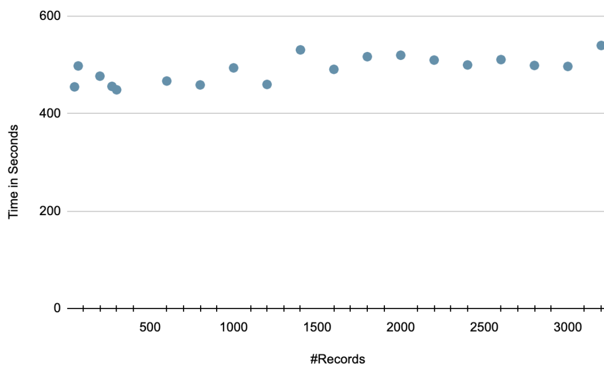

FIGURE 19. Number of records (intermediate key-value pairs) vs. time.

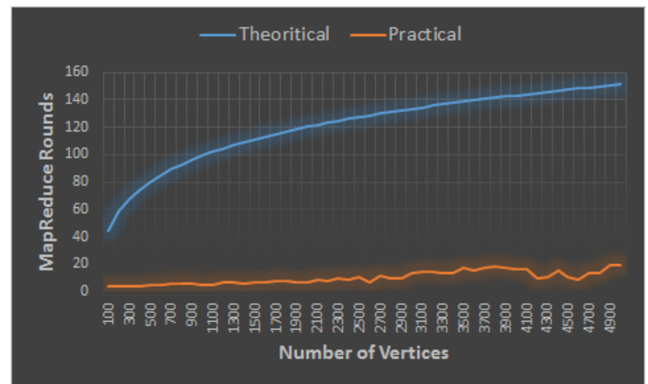

FIGURE 21. Number of vertices vs. MapReduce iterations (worst case).

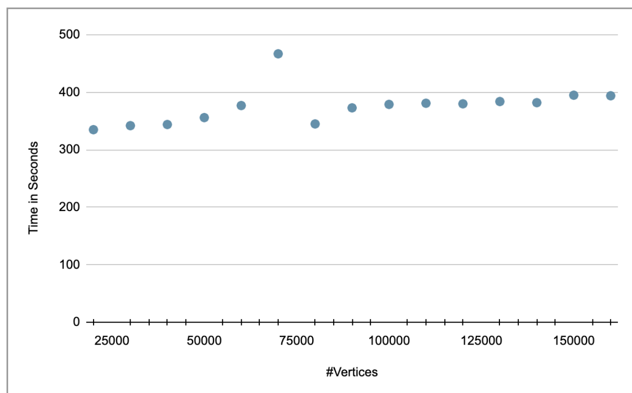

FIGURE 20. Vertices vs. time for large graph (max vertices 2 Million).

the node ID "value"). This approach works because the vertices represented by the key and value are still connected to some other vertices and remain in the graph. Therefore, we won't lose the endpoint vertices in this case, even if we drop a selected edge.

2) Dropping an edge from the edge list which connects vertices with degree one:
This is applicable when one of the vertices or both the vertices (end points a selected edge for dropping) have degree one. This case is not straightforward because the vertices involved in the process have degree one. By dropping the corresponding key-value pair of an edge, we will lose either one or both vertices from the graph's edge list (the vertex or vertices with degree one will be lost by dropping the corresponding $\langle key, value \rangle$ pair).
3) A bidirectional edge $\langle u, v \rangle$ can also be represented as $\langle v, u \rangle$. While generating an edge list from an adjacency list, such bidirectional edges create redundancy in the edge list. The redundancy causes the input graph size to be doubled because we work with undirected graphs. Removing the edge redundancy from the input graph reduced processing time dramatically.

For Figure 7 the edge list will be :

$$\langle 2, 1\rangle$$
$$\langle 3, 1\rangle$$
$$\langle 3, 2\rangle$$
$$\langle 4, 2\rangle$$

From the above edge list, if we drop an edge $\langle 3, 1\rangle$, simply removing that entry from the edge list is a sufficient task. However, if we drop an edge $\langle 4, 2\rangle$ from the above edge, we will lose vertex four from the new edge list. It will be like, vertex 4 was never in the given input graph.

B. SOLUTIONS

An additional MapReduce job is written for dropping an edge from the edge list, which will handle all the scenarios presented above while dropping an edge. For the vertices with degree one, an extra pair of keys and values is emitted with the key as vertex label and value as -1, showing it is a disconnected vertex from the rest of the input graph. For the example given in figure 7, while dropping an edge entry $\langle 4, 2\rangle$, a new entry in the new edge list is created with the record as $\langle 4, -1\rangle$. This shows that node Id 4 is disconnected in the new graph that is being generated.

VIII. FUTURE SCOPE

As an enhancement to this work, it would be an exciting topic of study to check how much time complexity differs after running the proposed algorithm separately on two different versions of the same graphs, 1. Dividing the original graph by a known bridge edge vs. 2. Removing a random edge (assuming it is a bridge) from the original graph. We expect some improvement in the performance on average in the former case, considering the graph gets divided into two halves which will reduce graph convergence time by half (the approximate best-case scenario).

Furthermore, we keep it as an open question whether it is possible to use distributed dynamic programming approach to improve performance considerably. In other words, to reuse computation done for checking if an edge is a bridge or not while checking the next edge for bridge property. Some of the calculations are repeated while finding bridges in the graph. The concept of Distributed Hash Table (DHT) [36] service can be utilized for storing possible repeated sub-computation. However, the question becomes challenging when we think that large and small star operations deal with one edge at a time. We cannot see beyond its neighbor in the distributed and parallel processing world (data chunks should be processed independently until the aggregation phase). Moreover, the large and small star Operations can add new edges and remove the original edges from the graph.

The proposed algorithm uses graphs generated by a graph generator. There are many open real-world graphs; for example, A semantic network [37], Twitter [38] etc., can be used as benchmarks. It would be an excellent enhancement to the results to do experiments with real-world graphs. The graph can be downloaded on the SNAP site (https://snap.stanford.edu/data/index.html).

Furthermore, the previous work for graph generation with real-world properties can be leveraged using synthetic graphs. There are many graph generators based on the MapReduce framework, for example, Power-Law Distributed Graph Generation With MapReduce [39], A Rapid and Robust Graph Generator [40], TrillionG: A trillion-scale synthetic graph generator using a recursive vector model [41], Distributed Tera-Scale Graph Generation and Visualization [42] etc.

IX. CONCLUSION

In this research, we proposed a new approach, BiECCA, for finding 2-edge connected components (2-ECCs) in parallel and distributed fashion for big-size graphs. Our work touches upon both graph theory and big data domains. We successfully solved one of the essential graph mining and analyzing problems on a big data analytics framework, Hadoop MapReduce.

We implemented the new distributed algorithm and showcased the correctness of the results, both theoretically and practically. We also analyzed the algorithm's time complexity and presented a lemma.

We then stated new ideas as a part of our work's future scope and enhancements. We encourage readers to explore these suggested ideas and open questions.

Finally, after comparing the practical and theoretical complexities of the proposed approach for finding 2-ECCs, it is evident that the proposed algorithm performs much better in real-world scenarios than predicted in the theoretical analysis. The proposed algorithm performs much better than the worst-case theoretical analysis.

ACKNOWLEDGMENT

The author would like to thank his mentor and advisor, Dr. Samuel Lomonaco, for guiding his research and encouraging him to pursue interesting problems. Dr. Samuel Lomonaco gave him the freedom to develop new ideas and has always encouraged him to pursue those. His keen insight and observations were very crucial to the progress of his research and completing it successfully. The author would also like to thank Dr. Christopher Marron and Dr. Ting Zhu for being external experts on his thesis defense committee. In addition, the author would like to thank his friend Anusha Dudi for brainstorming with him on various topics related to his work and for her critical comments and suggestions. Finally, the author is grateful to the University of Maryland Baltimore County for providing study resources, access to digital library resources, and study space available 24×7.

REFERENCES

[1] J. A. Bondy and U. S. R. Murty, *Graph Theory With Applications*. Amsterdam, The Netherlands: Elsevier, 1976.

[2] E. A. Kalinina and G. M. Khitrov, "A linear algebra approach to some problems of graph theory," in *Proc. Comput. Sci. Inf. Technol. (CSIT)*, Sep. 2017, pp. 5–8.

[3] W. Liang and B. D. McKay, "Fast parallel algorithms for testing k-connectivity of directed and undirected graphs," in *Proc. 1st Int. Conf. Algorithms Archit. Parallel Process.*, 1995, pp. 437–441.

[4] R. Kiveris, S. Lattanzi, V. Mirrokni, V. Rastogi, and S. Vassilvitskii, "Connected components in MapReduce and beyond," in *Proc. ACM Symp. Cloud Comput.*, Nov. 2014, pp. 1–13.

[5] A. Wigderson, "The complexity of graph connectivity," in *Mathematical Foundations of Computer Science 1992*, I. M. Havel and V. Koubek, Eds. Berlin, Germany: Springer, 1992, pp. 112–132.

[6] D. Dahiphale, R. Karve, A. V. Vasilakos, H. Liu, Z. Yu, A. Chhajer, J. Wang, and C. Wang, "An advanced MapReduce: Cloud MapReduce, enhancements and applications," *IEEE Trans. Netw. Service Manage.*, vol. 11, no. 1, pp. 101–115, Mar. 2014.

[7] N. Jasika, N. Alispahic, A. Elma, K. Ilvana, L. Elma, and N. Nosovic, "Dijkstra's shortest path algorithm serial and parallel execution performance analysis," in *Proc. 35th Int. Conv. MIPRO*, May 2012, pp. 1811–1815.

[8] R. Gentilini, C. Piazza, and A. Policriti, "Computing strongly connected components in a linear number of symbolic steps," in *Proc. SODA*, vol. 3, 2003, pp. 573–582.

[9] L. K. Fleischer, B. Hendrickson, and A. Pınar, "On identifying strongly connected components in parallel," in *Parallel and Distributed Processing*, J. Rolim, Ed. Berlin, Germany: Springer, 2000, pp. 505–511.

[10] R. Tarjan, "Depth-first search and linear graph algorithms," in *Proc. 12th Annu. Symp. Switching Automata Theory*, Oct. 1971, pp. 114–121.

[11] S. R. Kosaraju, "Efficient tree pattern matching," in *Proc. 30th Annu. Symp. Found. Comput. Sci.*, Nov. 1989, pp. 178–183.

[12] J. S. Vitter, "Algorithms and data structures for external memory," *Found. Trends® Theor. Comput. Sci.*, vol. 2, no. 4, pp. 305–474, 2008.

[13] M. Henzinger, S. Krinninger, and V. Loitzenbauer, "Finding 2-edge and 2-vertex strongly connected components in quadratic time," 2014, *arXiv:1412.6466*.

[14] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proc. OSDI*, 2004, pp. 137–150.

[15] *Hadoop*. Accessed: Dec. 2022. [Online]. Available: http://hadoop.apache.org/

[16] M. Luo, X. Yang, H. Zhang, Y. Zhang, L. Ji, and R. Li, "Construction and application of knowledge graph for bridge inspection," in *Proc. IEEE 10th Joint Int. Inf. Technol. Artif. Intell. Conf. (ITAIC)*, vol. 10, Jun. 2022, pp. 2565–2569.

[17] D. Dahiphale. (2017). *An Algorithm for Finding 2-Edge Connected Components in Undirected Graphs Using MapReduc*. [Online]. Available: https://www.proquest.com/dissertations-theses/algorithm-finding-2-edge-connected-components/docview/1940284806/se-2

[18] M. Posypkin and S. Thu Thant Sin, "Comparative performance study of shared and distributed memory dynamic programming algorithms," in *Proc. IEEE Conf. Russian Young Researchers Electr. Electron. Eng. (EIConRus)*, Jan. 2020, pp. 2010–2013.

[19] G. S. Bloom and S. W. Golomb, "Applications of numbered undirected graphs," *Proc. IEEE*, vol. 65, no. 4, pp. 562–570, Apr. 1977.

[20] T. Kamada and S. Kawai, "An algorithm for drawing general undirected graphs," Dept. Inf. Sci., Fac. Sci., Univ. Tokyo, Tokyo, Japan, Tech. Rep., 113, 1989.

[21] M. T. Thai and D.-Z. Du, "Connected dominating sets in disk graphs with bidirectional links," *IEEE Commun. Lett.*, vol. 10, no. 3, pp. 138–140, Mar. 2006.

[22] S. L. Hakimi, "On the degrees of the vertices of a directed graph," *J. Franklin Inst.*, vol. 279, no. 4, pp. 290–308, Apr. 1965. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0016003265903406

[23] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, "MapReduce online," in *Proc. NSDI*, 2010, pp. 313–328.

[24] R. Karve, D. Dahiphale, and A. Chhajer, "Optimizing cloud MapReduce for processing stream data using pipelining," in *Proc. 5th Eur. Symp. Comput. Modeling Simulation*, Nov. 2011, pp. 344–349.

[25] D. D. Mishra, S. Pathan, and C. Murthy. (2019). *Apache Spark Based Analytics of Squid Proxy Logs*. [Online]. Available: https://ieeexplore.ieee.org/document/8710044

[26] T. Oki, S. Taoka, and T. Watanabe, "A parallel algorithm for 2-edge-connectivity augmentation of a connected graph with multipartition constraints," in *Proc. 1st Int. Conf. Netw. Comput.*, Nov. 2010, pp. 227–231.

[27] H. Nagamochi and T. Ibaraki, "A linear-time algorithm for finding a sparse$k$-connected spanning subgraph of ak-connected graph," *Algorithmica*, vol. 7, nos. 1–6, pp. 583–596, Jun. 1992.

[28] L. Georgiadis and R. E. Tarjan, "Dominator tree certification and independent spanning trees," 2013, *arXiv:1210.8303*.

[29] R. Jaberi, "On computing the 2-vertex-connected components of directed graphs," *Discrete Appl. Math.*, vols. 164–172, pp. 41–44, Jan. 2016.

[30] Y. M. Erusalimskii and G. G. Svetlov, "Bijoin points, bibridges, and biblocks of directed graphs," *Cybernetics*, vol. 16, no. 1, pp. 41–44, 1980.

[31] D. W. Matula, "Determining edge connectivity in 0(nm)," in *Proc. 28th Annu. Symp. Found. Comput. Sci.*, Oct. 1987, pp. 249–251.

[32] H. Nagamochi and T. Watanabe, "Computing k-edge-connected components of a multigraph (special section on discrete mathematics and its applications)," *IEICE Trans. Fundamentals Electron., Commun. Comput. Sci.*, vol. 76, pp. 513–517, Jan. 1993.

[33] M. Hlawatsch, M. Burch, and D. Weiskopf, "Visual adjacency lists for dynamic graphs," *IEEE Trans. Vis. Comput. Graph.*, vol. 20, no. 11, pp. 1590–1603, Nov. 2014.

[34] T. A. Ayall, H. Liu, C. Zhou, A. M. Seid, F. B. Gereme, H. N. Abishu, and Y. H. Yacob, "Graph computing systems and partitioning techniques: A survey," *IEEE Access*, vol. 10, pp. 118523–118550, 2022.

[35] A. N. Nagiwale, M. R. Umale, and A. K. Sinha, "Design of self-adjusting algorithm for data-intensive MapReduce applications," in *Proc. Int. Conf. Energy Syst. Appl.*, Oct. 2015, pp. 506–510.

[36] T. Li, X. Zhou, K. Brandstatter, D. Zhao, K. Wang, A. Rajendran, Z. Zhang, and I. Raicu, "ZHT: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table," in *Proc. IEEE 27th Int. Symp. Parallel Distrib. Process.*, May 2013, pp. 775–787.

[37] M. Richardson, R. Agrawal, and P. Domingos, "Trust management for the semantic web," in *Proc. Int. Semantic Web Conf.*, Sanibel Island, FL, USA: Springer, Oct. 2003, pp. 351–368.

[38] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a social network or a news media?" in *Proc. 19th Int. Conf. World Wide Web*, Apr. 2010, pp. 591–600.

[39] R. Angles, F. López-Gallegos, and R. Paredes, "Power-law distributed graph generation with MapReduce," *IEEE Access*, vol. 9, pp. 94405–94415, 2021.

[40] R. Angles, R. Paredes, and R. García, "R3MAT: A rapid and robust graph generator," *IEEE Access*, vol. 8, pp. 130048–130065, 2020.

[41] H. Park and M.-S. Kim, "TrillionG: A trillion-scale synthetic graph generator using a recursive vector model," in *Proc. ACM Int. Conf. Manage. Data*, May 2017, pp. 913–928.

[42] B. Jeon, I. Jeon, and U. Kang, "TeGViz: Distributed tera-scale graph generation and visualization," in *Proc. IEEE Int. Conf. Data Mining Workshop (ICDMW)*, Nov. 2015, pp. 1620–1623.

DEVENDRA DAHIPHALE (Member, IEEE) received the Diploma degree in teacher education from the University of Pune, India, the B.A. degree in English from Yashvantrao Chavan Maharashtra Open University (YCMOU), Nashik, India, the B.E. degree in computer science and engineering from the Pune Institute of Computer Technology, India, and the M.Sc. degree in computer science from the University of Maryland, Baltimore County. He was a Software Engineer with Google, USA, Cisco Systems, USA, Amazon.com, USA, and Imagination Technologies, India. He is currently a member of Dreamz Group, a group of software professionals passionate about innovation and technology development. He has more than ten years of professional experience as a Software Engineer.

• • •