Fall 2022

CoDiS: Community Detection via Distributed Seed-Set Expansion on Graph Streams

Austin Anderson
San Jose State University

Recommended Citation
Anderson, Austin, "CoDiS: Community Detection via Distributed Seed-Set Expansion on Graph Streams" (2022). *Master's Projects*. 1190.
DOI: https://doi.org/10.31979/etd.fqw2-7bus
https://scholarworks.sjsu.edu/etd_projects/1190

CoDiS: Community Detection via Distributed Seed-Set Expansion on Graph Streams

A Project

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Austin Anderson

December 2022

The Designated Project Committee Approves the Project Titled

CoDiS: Community Detection via Distributed Seed-Set Expansion on Graph Streams

by

Austin Anderson

APPROVED FOR THE DEPARTMENTS OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

December 2022

Katerina Potika	Department of Computer Science
Robert Chun	Department of Computer Science
William Andreopoulos	Department of Computer Science

ABSTRACT

CoDiS: Community Detection via Distributed Seed-Set Expansion on Graph Streams

by Austin Anderson

Community detection has been and remains a very important topic in several fields. From marketing and social networking to biological studies, community detection plays a key role in advancing research in many different fields. Research on this topic originally looked at classifying nodes into discrete communities, but eventually moved forward to placing nodes in multiple communities. Unfortunately, community detection has always been a time-inefficient process, and recent data sets have been simply to large to realistically process using traditional methods. Because of this, recent methods have turned to parallelism, but all these methods, while offering significant decrease in processing time, still have several issues. The innovation of this paper is that it distributes the seed nodes instead of the stream edges, and therefore assigns to each working node a subset of the current formed communities. Experimental results show that we are able to gain a significant improvement in running time with no loss of accuracy.

ACKNOWLEDGMENTS

I would like to thank my advisor, Professor Katerina Potika, for her constant support and assistance, without which this project would never have been possible. Her accurate advice and deep knowledge of the subject were indispensable, and I greatly appreciate her time and investment in helping me with this project.

I would also like to thank Professor Robert Chun and Professor William Andreopoulos for graciously agreeing to be on my committee and giving me their time to support my project.

I would like to thank Dr. Liakos, who was kind enough to give me some pointers when I had difficulties reproducing his results.

Finally, I would like to thank my family, who has supported me throughout my entire academic career, and long before that as well.

TABLE OF CONTENTS

CHAPTER

1 Introduction 1

1.1 Problem Definition 3

1.2 Paper Structure 3

2 Terminology 5

3 Existing Methods 11

3.1 Early Methods 11

3.1.1 Girvan-Newman 11

3.1.2 Louvain 12

3.2 Later Approaches 12

3.2.1 CONGA and GONGO 13

3.2.2 CoEuS 13

3.2.3 Label Propagation 14

3.3 Current Parallel Methods 14

3.3.1 Subgraphs 15

3.3.2 DiCeS 16

4 New Methods 17

4.1 CoDiS 17

4.1.1 Splitting the Communities 17

4.1.2 Edge Ingestion and Distribution 19

4.1.3 Edge Processing and Community Pruning 20

4.1.4 F1 Score Collection and Termination 22

4.1.5 Apache Storm 24

5 Experimental Results 25

5.1 F1 Score Evaluation 26

5.2 Execution Time Evaluation 27

5.3 Space Usage Evaluation 28

5.4 Issues with Non-Shuffled Input Graphs 30

6 Results and Conclusion 35

6.1 Future Work 35

6.1.1 Custom `SortedMap` 36

6.1.2 Parallelize Pruning 36

LIST OF REFERENCES 37

APPENDIX

LIST OF FIGURES

1 Figure 6 from [1], shows a graph of paper references that has been partitioned, revealing communities of papers with similar subjects 1

2 Context of how CoDiS distributes the work to several worker nodes 4

3 an example of a graph with 10 nodes 5

4 an example of an adjacency matrix for a graph with 10 nodes . . 6

5 an example of an adjacency list for a graph with 10 nodes 7

8 an example of an Apache Storm topology in which there is one spout, 3 bolts that listen to the output of that spout, and two additional bolts that listen to the output of the 3 previous bolts . 10

9 An example of how the Girvan-Newman algorithm works 12

10 Figure 2 from [2] that shows how a node is split into two copies of itself and the edges from the original node are distributed to the new nodes 13

11 Figure 1 from [3], which shows an example of a graph split into subgraphs that communicate with each other 15

12 Figure 1 from [4], shows how the edges are distributed to individual workers that all access shared data 16

13 Comparison of average F1 scores for DiCeS [4] and CoDiS, averaged over 10 runs 27

14 Comparison of average time per edge for CoDiS and DiCeS when run on our test machine with 2, 3, and 4 processing nodes, averaged over 10 runs 28

16 The edge times reported by [4] from testing on their machine using 4 processing bolts 31

17 Comparison of the expected DiCeS [4] time per edge vs the actual time per edge on the YouTube dataset with 4 bolts 32

18 Comparison of DiCeS [4] running times observed after shuffling the input graph, as opposed to the expected and observed times from the non-shuffled input graph 33

CHAPTER 1

Introduction

Community detection is crucial technique for many different areas of interest from biology to social network analysis and even criminal justice, any time it is useful to find groups of individuals from a seemingly arbitrarily connected network, community detection algorithms are employed. Research on community detection has been ongoing for several decades, with new methods being researched to increase the accuracy, speed, and robustness of community detection algorithms.

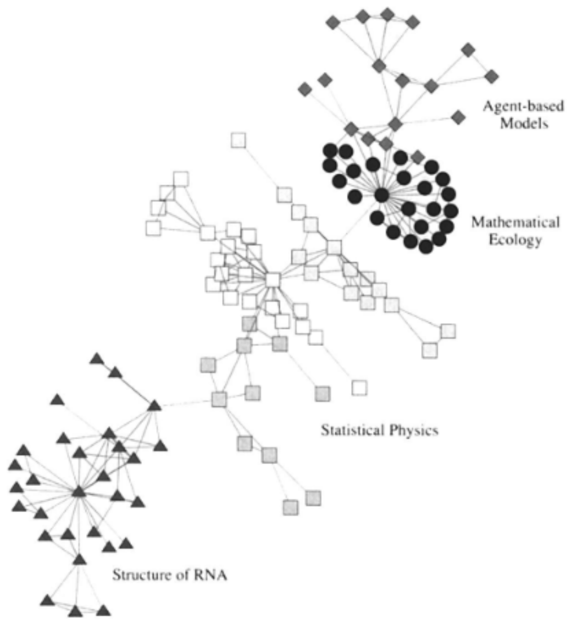


Figure 1: Figure 6 from [1], shows a graph of paper references that has been partitioned, revealing communities of papers with similar subjects

Unfortunately, as the internet has continued to grow, and our ability to collect data has improved, the size and complexity of the networks we wish to analyze has become prohibitively large. Social networking sites like Facebook or Twitter have millions of daily users each, all friending each-other or mentioning each-other in tweets and posts several million times a day, which creates massive, complex networks great for social network analysis. Amazon alone has several hundred million products for sale, and finding which customers tend to buy similar items is crucial for effectively marketing to those groups. Because of this it has become difficult to not only process these networks in a realistic timeframe, but even just storing and accessing them in a timely manner has become a challenge.

Because of this, much of the recent research on community detection has been focused not on increasing the accuracy of community detection algorithms, but instead on increasing the speed and amount of data that can be processed. Along those same lines, community detection algorithms that can take advantage of new multi-threaded CPUs and distributed computing have also become the topic of more and more research papers.

One of these papers is CoEuS [5], by Liakos et al. [5] is a unique approach to the community detection problem in that it does not try to partition the *entire* graph into communities, but instead looks at only a select number of communities of interest. This works by providing a small set of known nodes from a community of interest, and then trying to build the rest of the community from that seed set of nodes. This allows Liakos et al. to look at the graph edge-by-edge instead of trying to load the entire thing at once, meaning much larger networks can be processed than in other approaches. In addition to this, Liakos et al. expanded on their approach with DiCeS [4], which uses Apache Storm and Redis to accelerate the algorithm proposed in [5] using distributed computing.

1.1 Problem Definition

While [4] is great for quickly analyzing large networks, it is not without issue. The way that [4] distributes the work is by having each worker node process edges in parallel, and write those edges to a shared list of communities, which is stored in a Redis cluster so that all worker nodes have access to all the communities that are in consideration. Because of this, there is added overhead to the worker nodes in accessing the community data from the Redis cluster, as well as concurrency protection increasing overhead to read and write data to the cluster.

Because of this, we propose a new method of parallelizing the algorithm of [5]. While [4] splits the edges among all the worker nodes, which then look at all the communities to determine if the nodes in that edge belong in any of those communities, our proposed method instead splits the *communities* among the worker nodes, so that each worker node is only looking at a subset of all the communities that are in consideration. By doing so, although we decrease the edge processing throughput, we also decrease the amount of time each worker node spends on each edge. Crucially, we also remove the need for every worker node to have access to every community in consideration, meaning we can remove Redis from the equation, regaining the time that was lost to concurrency protection and data distribution

1.2 Paper Structure

The rest of this paper is structured as follows: immediately following this introduction, there is a list of terms that are useful to know when reading this paper. Next, Chapter 3 will explore several current community detection approaches, from well know and oft-cited approaches to recent papers focusing on parallel community detection approaches. After that, chapters 4 and 5 will go in-depth into the changes

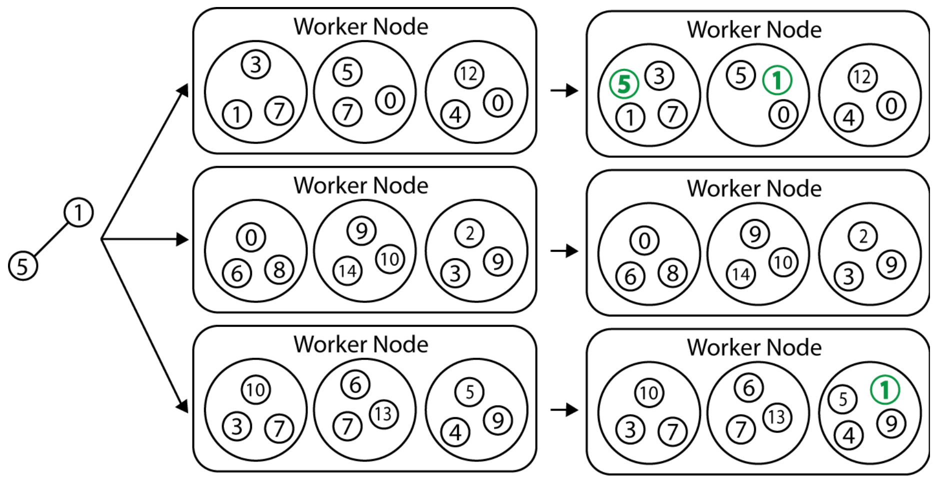


Figure 2: Context of how CoDiS distributes the work to several worker nodes

we are proposing and the results of the experiments related to those changes. Finally, chapter 6 will wrap everything up and discuss more changes that we would like to consider for future work

CHAPTER 2

Terminology

In this chapter, we have the basic definitions and notations.

Graph: A graph is a set of vertices and edges $G = (V, E)$, where V is the set of all vertices in the graph and E is the set of all edges. Graphs are very versatile and can be used to represent any number of complex ideas in an easy-to-parse data structure, from geographical layouts to complex interactions in social media networks.

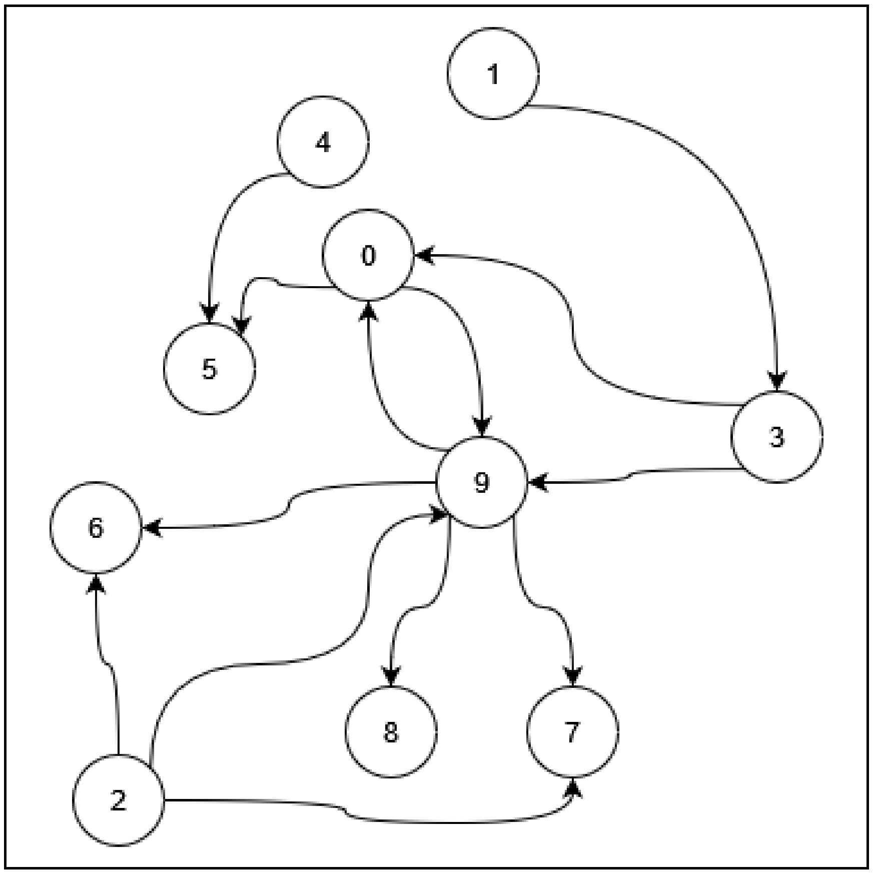
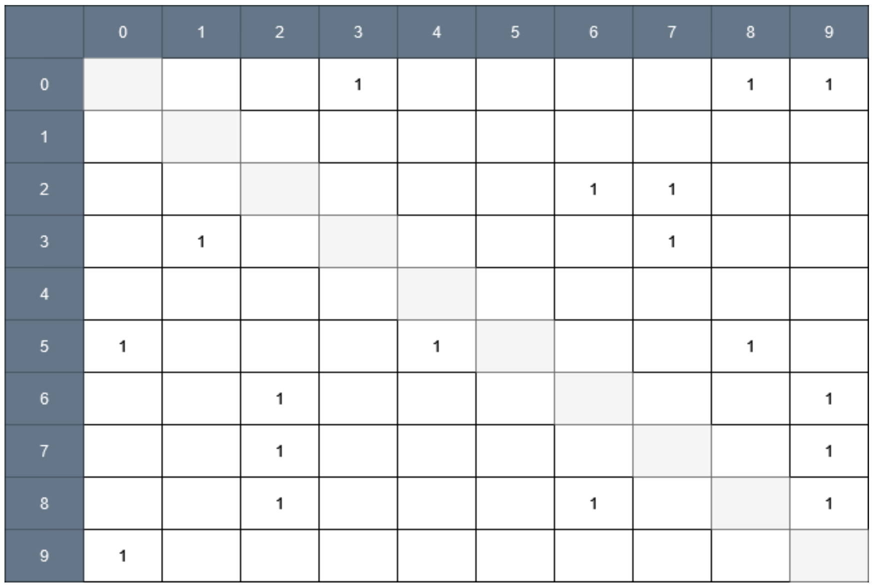


Figure 3: an example of a graph with 10 nodes

Adjacency Matrix: A method of representing a graph in which the graph is stored as a NxN matrix A, where N is the number of nodes in the graph. The value at $A[x][y]$ represents the presence or absence of an edge from node x to node y. For an unweighted graph, A will be a matrix of `bool` values, with a value of *true* meaning an edge exists. For a weighted graph, A can be a matrix of something like `integer` or `double`, with a $NULL$ value meaning there is no edge, and any other value being the weight of the edge from x to y. This method of graph representation allows quick lookup of edges, but also requires a large amount of memory to store, with a space complexity of N^2.



	0	1	2	3	4	5	6	7	8	9
0				1					1	1
1										
2							1	1		
3		1						1		
4										
5	1				1				1	
6			1							1
7			1							1
8			1				1			1
9	1									

Figure 4: an example of an adjacency matrix for a graph with 10 nodes

Adjacency List: A method of representing a graph in which the graph is represented as an array of lists of nodes. Each node in the graph has its own list. For any given node x, if its associated list contains the node y, that means there is an edge from node x to node y. This method of graph representation is much more space efficient than an Adjacency Matrix, scaling linearly with the number of edges. On the other hand, the time to look up an edge is much longer, N in the worst case and scaling with N as well as how densely the graph is interconnected.

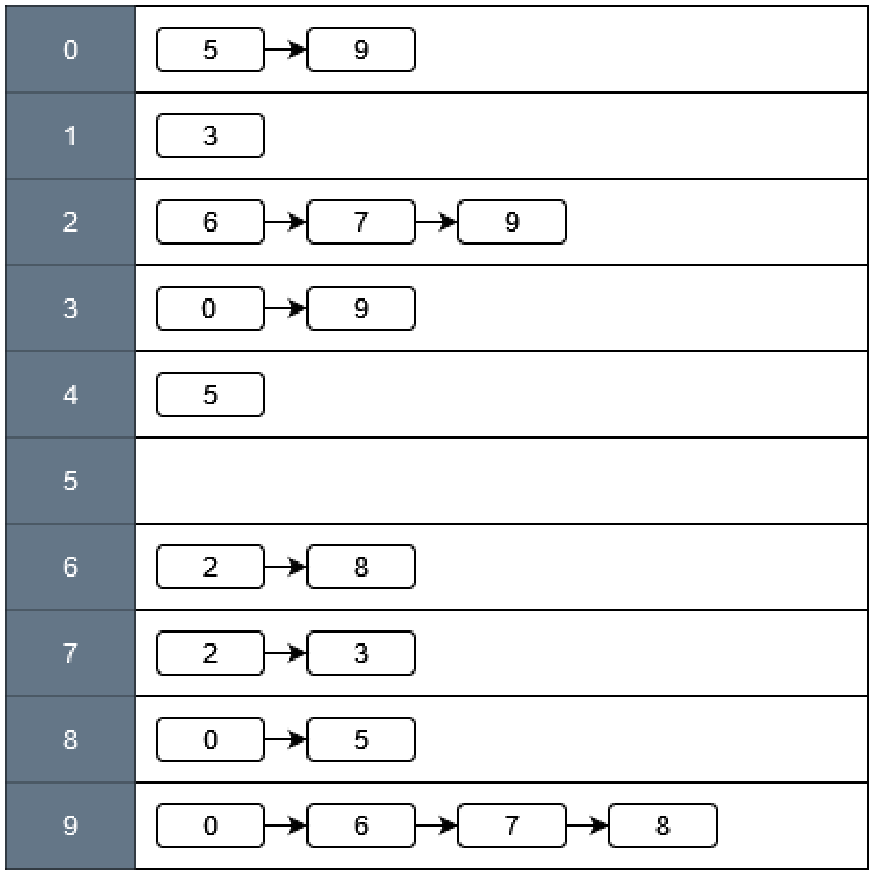


Figure 5: an example of an adjacency list for a graph with 10 nodes

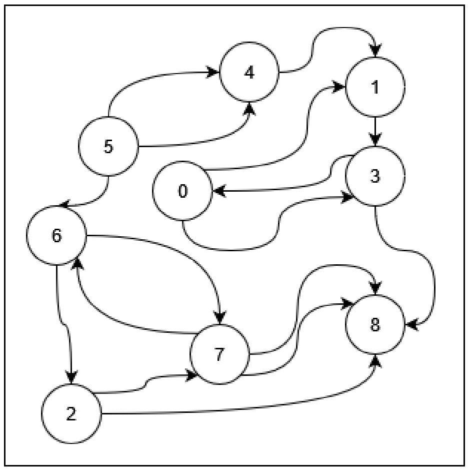
Node/Vertex: A vertex, also called a node, is one of the two basic building blocks of a graph. A vertex represents objects, but what those objects are changes depending on the graph. For example, a vertex might represent a person in a graph that describes a social network, a product in a graph that describes a shopping website, or cities in a graph that describes a map. Depending on the graph, a vertex could represent something more complex or abstract, such as a grouping of objects, or even other graphs.

Edge: The other of the two basic building blocks of a graph, edges represent connections between up to two vertices. An edge can connect a vertex to itself or another vertex, but a single edge cannot connect more than two vertices. Edges can have several additional properties, such as weights or a direction, but these are not required.

Subgraph: A subgraph is a graph made of a subset of the vertices and edges of another graph. Given a graph $G = (V, E)$ and a subgraph of G $G' = (V', E')$, then both $\forall v \in V', v \in V$ and $\forall e \in E', e \in E$ are true.

Community: A community is a subset of vertices from a graph that are densely connected to each other and weakly connected to the rest of the graph.

Community Detection: Community detection is the process of trying to group vertices of a graph into communities.



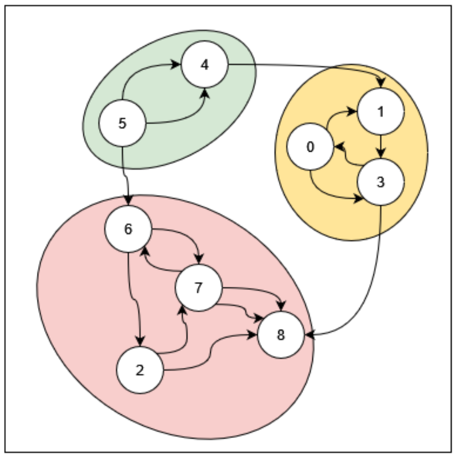


Figure 6: Left: a graph before being split into communities
Right: a graph with found communities highlighted

Edge Betweenness: A measure of how commonly traversed an edge is when finding the shortest path from every vertex in a graph to every other vertex. The edge betweenness of an edge e is calculated as the number of shortest paths that include edge e over the total number of shortest paths

Modularity Score: An approximation of how close the found communities match an ideal community distribution. The higher the modularity score, the closer the graph is to being separated perfectly into communities, and vice versa

Participation Score: A calculation of how strongly connected a given graph node is to the nodes in a given community.

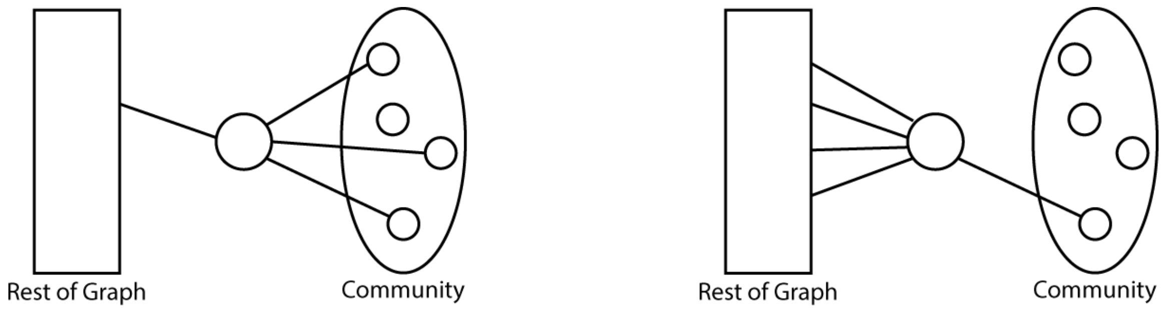


Figure 7: Left: a node with high participation score for the community
Right: a node with low participation score for the community

Apache Storm: A java framework for distributed processing using streams that is maintained by Twitter and the Apache Foundation. Storm abstracts the distribution and passing of messages between compute nodes. The basic datatype of a Storm distribution is a `Tuple`, and there are two types of processing nodes: `Spout`s and `Bolt`s.

Tuple: The basic datatype of an Apache Storm distribution. A `Tuple` contains any number of supported data-types. `Tuple`s natively support all primitive data types, as well as `String`s and `byte` arrays. `Tuple`s can also support custom data types if those data types implement a serializer and register that serializer with Storm.

Topology: A `Topology` is the construct that contains all the information about the particular setup for a given Storm program. The `Topology` is what is submitted to an Apache Storm cluster and contains the code that computation nodes execute, as well as how the messages are distributed between nodes and which nodes listen to each other. A `Topology` will run indefinitely until the user stops it or submits a new topology to the Storm cluster.

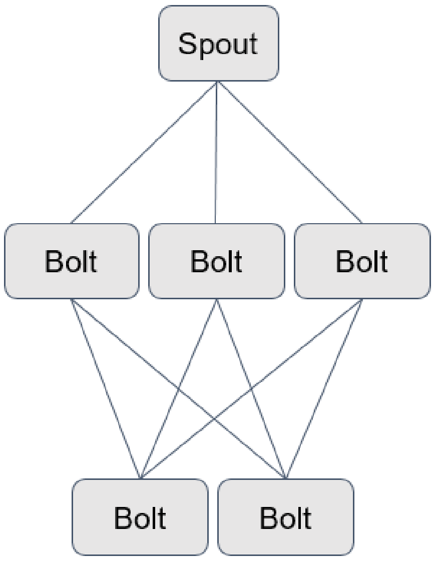


Figure 8: an example of an Apache Storm topology in which there is one spout, 3 bolts that listen to the output of that spout, and two additional bolts that listen to the output of the 3 previous bolts

Spout: One of the two main components of an Apache Storm `Topology`. A `Spout` is responsible for creating a stream of `Tuple`s that are then emitted to the rest of the topology. The `Topology` itself takes care of where how those `Tuple`s are distributed. `Spout`s cannot receive `Tuple`s from any other processing nodes, only create and emit them.

Bolt: One of the two main components of an Apache Storm `Topology`. `Bolt`s are typically the nodes that do the actual processing of the topology. `Bolt`s receive the `Tuple`s emitted from `Spout`s or even other `Bolt`s, and do some kind of processing on them. `Bolt`s can emit their own `Tuple`s for consumption by other `Bolt`s.

Apache Storm Local Mode: A way of running an Apache Storm cluster that simulates distributed computing by using multi-threading instead of multiple systems

CHAPTER 3

Existing Methods

3.1 Early Methods

Many of the early community detection algorithms were very simple and restrictive, but many of the concepts are still being used in modern approaches. Before these methods, simple max-flow/min-cut approaches were the norm [6], but these papers introduced concepts that revolutionized the way community detection was approached

3.1.1 Girvan-Newman

The Girvan-Newman [1] algorithm was one of the community detection algorithms to gain popularity. The paper introduces two major concepts that are still used in many modern approaches: Edge-Betweenness and Modularity score. Edge-Betweenness is used to determine how strongly or weakly an edge is tied to the rest of the graph.

Then, using the observation that an edge that has high Edge-Betweenness is likely a connection between communities, the algorithm calculates the edge-betweenness of every edge in the graph, and then removes the edge with the greatest edge-betweenness. This process continues until it is determined that the graph has been properly partitioned into communities, and in the case of [1], the modularity score of the graph is calculated every time an edge is removed, and once the modularity score stops increasing, the algorithm finishes.

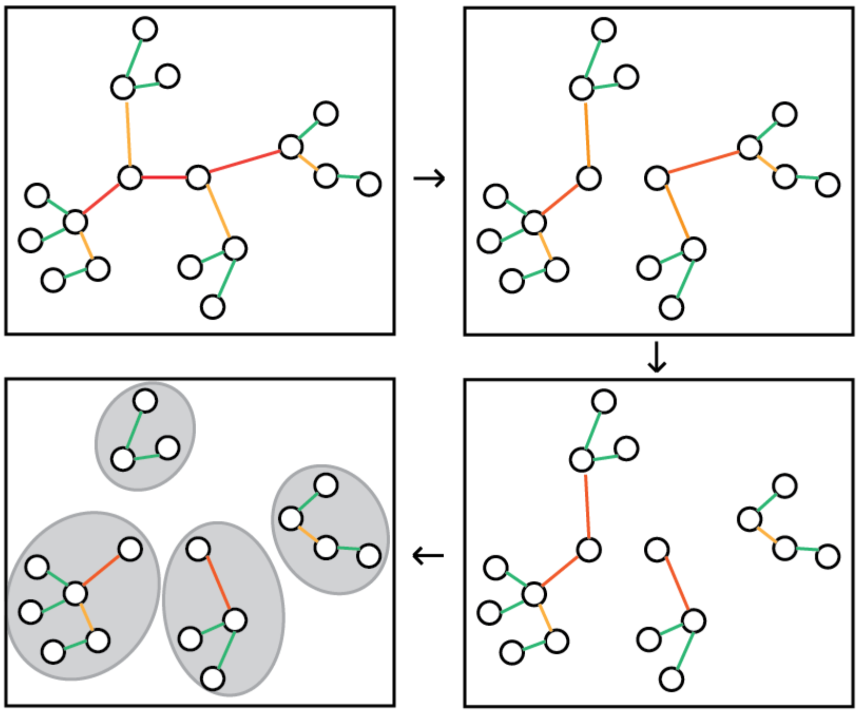


Figure 9: An example of how the Girvan-Newman algorithm works

3.1.2 Louvain

The Louvain method [7] takes the modularity score from Girvan-Newman and makes it the focus of the algorithm. [7] begins with every node of the graph belonging to its own community. Then, for every node and every possible community that node could be a part of, calculate the change in the modularity score that would occur from moving that node to that community. The change that would bring about the greatest increase in modularity score is then applied, and this continues until there are no changes that can be made that would increase the modularity score

3.2 Later Approaches

While the earlier methods of community detection are incredibly important, they were also slow and restrictive. For example, in both the [1] and [7], nodes could only belong to a single community, which is unrealistic for real-world applications. The following approaches attempt to solve one or more of these issues.

3.2.1 CONGA and GONGO

CONGA [2] and GONGO [8] are both variants of the Girvan-Newman algorithm that allow nodes to participate in more than one community. Instead of calculating which edge has the greatest edge-betweenness and removing it, these algorithms calculate which node has the greatest node-betweenness and splits that node. The edges of the original node are distributed to the copy nodes, and when the algorithm finishes the communities a node belongs to are all the communities its copies are part of.

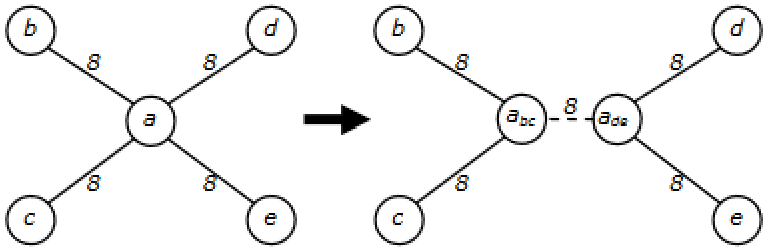


Figure 10: Figure 2 from [2] that shows how a node is split into two copies of itself and the edges from the original node are distributed to the new nodes

3.2.2 CoEuS

CoEuS [5] attempts to solve the speed problem by only looking at a part of the graph. This method works by starting with a seed set of communities, then streams in pairs of nodes that are connected by an edge. These nodes are placed in communities based on what communities those nodes are already in, or if neither of those nodes has been seen yet, a new community is created that is just those two nodes.

3.2.3 Label Propagation

Label propagation algorithms work by giving nodes labels, and then spreading those labels around. It is very easy to make these algorithms applicable to overlapping community detection by simply allowing every node to maintain a list of labels, instead of having only a single label. Once the algorithm ends, the communities that any node belongs to are the labels that exist in that nodes list of labels.

3.2.3.1 COPRA

COPRA [9]works by having each node look at all the nodes of its neighbors, then take the label that is most prevalent and adds it to its list of nodes. If there is a tie it is broken in a random but deterministic way.

3.2.3.2 SLPA

SLPA stands for Speaker-Listener Label Propagation Algorithm [10]. In this algorithm, instead of each node looking at its neighbors' labels, nodes choose a random label from their list of labels and broadcast it to all of their neighbors. Although randomly selected, the choice is weighted, so labels that have been seen often are more likely to be broadcast than those rarely seen.

3.3 Current Parallel Methods

With the advent and proliferation of multi-core CPUs and the increased access to distributed computing, parallelization has become a new focus in the community detection space as a way to further increase the speed and data capacity of community detection algorithms.

3.3.1 Subgraphs

Many parallel implementations work by taking a non-parallel algorithm and having it run on subsections of the graph. These subsections are then recombined in some way to get the full community graph structure. The way these approaches split the graph into subgraphs, and how the subgraphs are recombined has a large effect on the outcome.

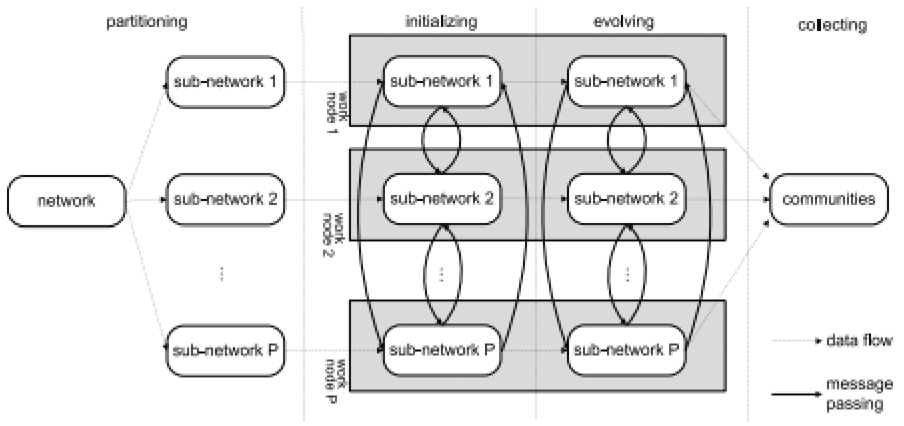


Figure 11: Figure 1 from [3], which shows an example of a graph split into subgraphs that communicate with each other

For example, [11], [12] and [3] all use a min-cut method to split the graph, but how they recombine the graphs is very different. When splitting the graph, [11] creates copies on any nodes that have edges to nodes that are part of other subgraphs, and those nodes are simply combined together when the algorithm is finished. While [12] also has a combining step at the end, it does not create duplicate nodes, and instead has to reconcile all the subgraphs based on their connections before separating the graph. Finally, [3] does not have a dedicated combination step like [11] or [12], but instead has a mini combining step at the end of every iteration of the algorithm.

While CoDiS does not use subgraphs directly, it is the concept of subgraphs that inspired the alternate method of distribution we employ.

3.3.2 DiCeS

DiCeS [4] is a distributed version of [5], using Apache Storm. It works by having several worker processes that can take in streamed nodes, and then decides where to put them in the community graph. The created community graph is accessible to each worker process.

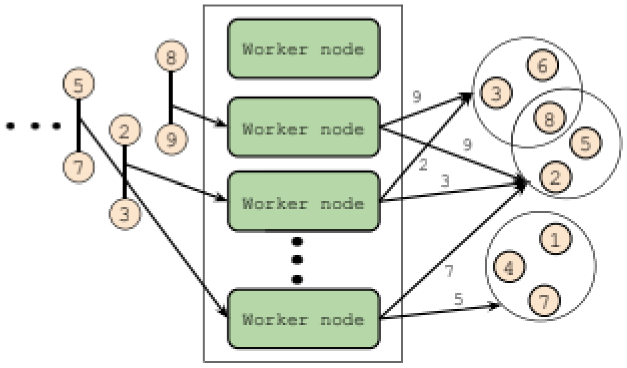


Figure 12: Figure 1 from [4], shows how the edges are distributed to individual workers that all access shared data

Improving this method is the main focus of our new approach. While [4] achieves speedups by parallelizing the edge processing using distributed computing, we are interested in trying to speed up processing time by splitting the search space between the different compute nodes. Chapter 4 will go into this topic in more detail.

CHAPTER 4

New Methods

4.1 CoDiS

Here we describe CoDiS, a new method that functions similarly to DiCeS in that it runs the CoEuS algorithm in a distributed manner to achieve parallelism, but modifies it so that each individual worker node only operates on a subset of the communities of interest. The implementation of this method is done in Java using Apache Storm to manage the distributed processing. Similar to DiCeS and CoEuS, this method requires a seed set of nodes for each community of interest, and the community detection is done by growing that set using participation score as a metric.

4.1.1 Splitting the Communities

The first step of CoDiSis to initialize the communities of interest and distribute them among the worker nodes. This is done by either explicitly providing a list of seed sets or by providing ground truth communities for the graph being processed and choosing a set of random nodes to be used as the seed set. Algorithm 1 shows the psuedocode for this process.

The process begins by recording the start time and receiving a list of ground truth communities (or seed sets, if the algorithm is being run on a dataset without ground truth communities available), as well as a list of identifiers to access specific worker nodes. Then, a copy of the list of worker nodes is created, but with the order of those nodes randomized. The next step is to initialize the communities and distribute them to to worker nodes

Algorithm 1: Community Initialization and Distribution

```
Input($G, T$) such that $G$ is a list of ground truth communities and $T$ is a set
  of worker node identifiers
begin
    $CollectionNode.startTime \longleftarrow$ Now()
    $shuffled \longleftarrow$ Shuffle($T$)
    $i \longleftarrow 0$
    for $g \in G$ do
        $S \longleftarrow \emptyset$
        $C \longleftarrow$ CoDiSCommunity()

        $C.seedSet \longleftarrow$ Set($String$)
        $C.nodes \longleftarrow$ Map($String$, $Double$)
        $C.commDegrees \longleftarrow$ Map($String$, $Double$)

        while len($S$) $< SEED\_SET\_SIZE$ do
            Append($S$, $g$[Rand()])
        end

        $C.seedSet \longleftarrow S$

        if $i \geq$ len($shuffled$) then
            $i \longleftarrow 1$
        end

        $shuffled[i++]$.AddToExecuteQueue($C$)
    end
end
```

This is done by creating a counter i that is initialized to zero, then iterating through all the ground truth communities. For each community, we initialize a `CoDiSCommunity` object. We then select a number of random nodes from the ground truth communities to be used as a seed set and add those to the `CoDiSCommunity` seed set list. Finally, we assign this community to the worker node at index i of our shuffled list, then increment i or reset it to 0 if it becomes greater than the length of the shuffled list.

4.1.2 Edge Ingestion and Distribution

The next step of the process is to start reading edges and distributing them to the worker nodes. The edges could be read from several sources, such as crawling the web to find links between pages or accessing a social media API to find connections between people, but for the purposes of our experiments, we find edges by reading from a text file.

Algorithm 2: Edge Ingestion and Distribution

```
1  Input($path, T$) such that $path$ is the filepath to the text file that contains the
     community to be read and $T$ is a set of worker node identifiers
2  begin
3      $FILE \longleftarrow$ Open($path$, "$r$")
4      $line \longleftarrow FILE$.NextLine()
5      while $line \neq EOF$ do
6          $s \longleftarrow line$.Split(" ")
7          $e \longleftarrow$ Tuple($s[0]$, $s[1]$)
8
9          for $t \in T$ do
10             $t$.AddToExecuteQueue($e$)
11         end
12         $line \longleftarrow FILE$.NextLine()
13     end
14     $t$.AddToExecuteQueue("$EOF$")
15 end
```

The process begins by taking as input a path to the file to read edges from as well as a list of identifiers for all the worker nodes. The file at *path* is opened for reading, and the process begins reading the file line-by-line. For every line, as long as the line isn't an *EOF* indicator, the process splits the line into the two node identifiers of the edge the line describes, and creates a `Tuple` object containing the two node identifiers. The process then adds the edge to the processing queue of all the worker threads, reads the next line from the file, and repeats the process. Once the *EOF* indicator is encountered, the process adds this to the processing queue of the worker nodes and finishes executing.

4.1.3 Edge Processing and Community Pruning

Algorithm 3 is where the bulk of the community detection process is done. This algorithm is run in parallel by all of the worker nodes. Each worker node maintains a queue of inputs received by the edge distribution node, and is simply a loop of popping the next input from the queue and running algorithm 3 with the input from the queue as the input to the algorithm

The first thing algorithm 3 does is check the datatype of the input that is received. If the input is a `String`, then we know the EOF indicator has been reached, and we can calculate the average F1 score of all the communities and pass that on to the collection node. Before calculating the F1 score, we prune the community to its final size, which can be done either by pruning it to the size of the ground truth community this community is based on, or by using the drop tail technique of [5].

If the input is a `CoDiSCommunity`, then that community is appended to the list of communities that the worker node has under inspection. If the input is a `Tuple`, then it is an edge that we need to process.

We begin by getting *nodeU* and *nodeV* from the `Tuple` and incrementing the total degrees for both of the nodes, as well as the counter that keeps track of how many edges have been processed in total by this worker node. Next, we find all the communities of this worker node that either of the nodes belong to and start iterating over all those communities. If the community contains *nodeU*, then we update the estimated community degrees of *nodeV* using the participation score of *nodeU*, then we add *nodeV* to the list of nodes that are part of this community. The same is then done for *nodeV*, updating and adding *nodeU* to the community if applicable.

Algorithm 3: Edge Processing

```
Input($T$) such that $T$ is either a String, a CoDiSCommunity, or a Tuple
begin
    switch Typeof($T$) do
        case String do
            $sumF1 \longleftarrow 0$
            for $C \in this.communities$ do
                $C$.Prune(GetSizeDetermination($C$))
                $CollectorNode$.AddToExecuteQueue(CalculateF1Score($C$))
            end
            $CollectorNode$.AddToExecuteQueue($this.numEdges$)
        end
        case CoDiSCommunity do
            Append($this.communities$, $T$)
        end
        case Tuple do
            $this.numEdges{++}$
            $nodeU \longleftarrow T[0]$
            $nodeV \longleftarrow T[1]$

            $this.degrees[nodeU]{++}$
            $this.degrees[nodeV]{++}$

            $commsU \longleftarrow$ GetCommunitiesContaining($nodeU$)
            $commsV \longleftarrow$ GetCommunitiesContaining($nodeV$)

            for $C \in (commsU \cup commsV)$ do
                if $C$.Contains($nodeU$) then
                    $C.commDegrees[nodeV] \mathrel{+}= \frac{C.commDegrees[nodeU]}{this.degrees[nodeU]}$
                    $C.nodes$.Put($NodeV$, $\frac{C.commDegrees[nodeV]}{this.degrees[nodeV]}$)
                end
                if $C$.Contains($nodeV$) then
                    $C.commDegrees[nodeU] \mathrel{+}= \frac{C.commDegrees[nodeV]}{this.degrees[nodeV]}$
                    $C.nodes$.Put($NodeU$, $\frac{C.commDegrees[nodeU]}{this.degrees[nodeU]}$)
                end
                if $this.numEdges \% PRUNE\_WINDOW == 0$ then
                    $C$.Prune($MAX\_COMMUNITY\_SIZE$)
                end
            end
        end
    end
end
```

Once the processing of the edge is done, we check if the number of processed edges is a multiple of the chosen pruning window, set to 10000 edges as per [4]. If it is a multiple, we use algorithm 4 to prune the community to a given size. This starts by getting a list of key-value pairs of the nodes the community contains and sorting that list in descending order by value. We then get a subset of this list with only the top $MAX\_COMMUNITY\_SIZE$ pairs and transform the sub-list back into a map, replacing the old community nodes map.

Algorithm 4: Community Pruning

```
1 Input($Size$) such that $Size$ is the number of nodes pairs that should remain
    in the community after pruning has been completed
2 begin
3     $sorted$ ⟵ GetPairsAsList($this.nodes$)
4     $sorted$ ⟵ SortByValueDescending($sorted$)
5     $sorted$ ⟵ $sorted$.Sublist($0$,$Size$)
6
7     $this.nodes$ ⟵ ListToMap($sorted$)
8 end
```

4.1.4 F1 Score Collection and Termination

Algorithm 5 is the process that receives the output from all the worker nodes, reports the results, and terminates the program. The process receives as input a `Tuple` from the worker nodes that contains either a `double` or a `integer`.

If the received `Tuple` is a `double`, then we know the process has received an F1 score for a single community from a worker node. We then add this F1 score to the total sum of F1 scores and increment the community counter.

If the received `Tuple` is an `integer`, we know that the worker node that sent it has finished calculating and sending the average F1 score for all the communities that that worker node has under consideration, and the integer sent is the total number of edges the worker node processed. When we receive an `integer`, we increment the finished workers counter, and when we have received as many as there are worker nodes, we can calculate and log the analysis data. We subtract the start time from the current time to get the total execution time, and then divide that by the number of edges to get the average time per edge. Finally, we divide the summed F1 scores by the total number of communities to get the average F1 score and write all these values to a log for analysis.

Algorithm 5: Collection and Termination

```
Input($T$) such that $T$ is a Tuple that contains either and double or a
  integer
begin
    switch Typeof($T$) do
        case double do
            $this.communityCounter++$
            $this.totalF1 += T$
        end
        case integer do
            $this.numFinishedWorkers++$
            if $this.numFinishedWorkers == TOTAL\_WORKERS$ then
                $executionTime \longleftarrow$ Now() $- this.startTime$
                $timePerEdge \longleftarrow \frac{executionTime}{T}$
                $averageF1 \longleftarrow \frac{this.totalF1}{this.communityCounter}$

                Log("Average F1: ", averageF1)
                Log("Total Time: ", executionTime)
                Log("Time per Edge: ", timePerEdge)

                Exit()
            end
        end
    end
end
```

4.1.5 Apache Storm

Just as in [4], Apache Storm is used to implement the distributed computing components of CoDiS. Apache Storm takes care of passing the messages between different execution nodes. In addition to handling communication between execution nodes, the Apache Storm implementation also contains a `Topology`, which handles how messages should be distributed between nodes, and which nodes receive messages from other nodes. For example, we do not explicitly distribute messages as in lines 4 and 20-24 of algorithm 1. Instead, once the `Topology` is set up, we simply call `emit(Tuple)`, and the `Topology` takes care of evenly distributing the emitted `Tuple`s to the worker nodes that are listening to the node emitting the `Tuple`s.

Algorithms 1 and 2 are handled by `CoDiSSpout`, which extends `Spout`. Algorithms 3 and 4 are handled by `CoDiSBolt`, which extends `Bolt`. Algorithm 5 is handled by `CoDiSCollectionBolt`, which also extends `Bolt`.

The `CoDiSSpout` is the start of the process, and therefore there is only one instance of it, which does not listen to the output of any other nodes. `CoDiSBolt` implements the worker nodes, so there are as many instances of `CoDiSBolt` as we want execution threads, and all the `CoDiSBolt`s listen to the output of the single `CoDiSSpout` instance. Finally, there is only a single `CoDiSCollectionBolt`, which listens to the output of all the `CoDiSBolt`s.

CHAPTER 5

Experimental Results

The experiments for the new methods proposed in chapter 4 were run on 4 of the 6 data-sets that [4] was run on. The system used for experimentation is a Linux box running Ubuntu 22.04.1, with a core i7-12700K and 64 GB of RAM. All tests were run using 2, 3, and 4 bolts in local mode on the same system under the same conditions, and were all run 10 times using Maven and OpenTap to automate running the tests and collecting the results. The input graph text file was shuffled before each test and the seed-set for each ground truth community was chosen randomly. The tests were run using the top 5000 ground truth communities with greater than 20 participating nodes, just as in [4].

Dataset	Nodes	Edges	DiCeS [4] Avg F1	CoDiS Avg F1
Amazon	334,863	925,872	0.817920	0.800006
DBLP	317,080	1,049,866	0.409246	0.412375
Youtube	1,134,890	2,987,624	0.091300	0.086503
LiveJournal	3,997,962	34,681,189	0.573400	0.563260

Table 1: The count of nodes and edges for each dataset that was used, as well as the average F1 score for both DiCeS [4] andCoDiS when run on our test machine

The reason only 4 of the 6 datasets from [4] are used is that both DiCeS and CoDiS eventually ran out of memory on our test system when running the Orkut and Friendster datasets. These datasets, as well as the top 5000 ground-truth communities, are publicly available[1]. The code that is run to test [4] is from the authors' public repository[2] provided for creating reproducible results. No changes to the source code

[1]https://snap.stanford.edu/data/#communities

[2]https://github.com/panagiotisl/DiCeS/

were made, the sole exception being the hard-coded file names and number of edges for the reproducible tests.

5.1 F1 Score Evaluation

Figure 13, as well as columns four and five of table 1 show the comparison of the average F1 score of the results of running our implementation and [4] on the same data-sets and test machine.

The average F1 score results of CoDiS are ± 0.02 of the results from running the provided code for [4] for all datasets. On average, the average F1 score of CoDiS was lower than that of [4], with the greatest discrepancy being the results from the Amazon dataset, with a difference of 0.017914. This is not a rule, however, as the average F1 score of the DBLP dataset when run in CoDiS was actually higher than the average F1 score when run on [4].

An observation we made during testing is that there was significantly more variation in the average F1 score for [4] than for CoDiS. When run with a non-random seed-set and with the same input graph file, the was no observed variation in the average F1 score for CoDiS on any dataset. This is not true of [4] however, and we continued to observe variation in F1 score even after locking the seed-sets. We believe this is due to the fact that the pruning step of [4] is done in parallel to the worker node execution, and so due to different OS scheduling, it is possible for certain edges to be processed during or after the pruning step, therefore changing the results run-by-run. This is not an issue in CoDiS, as it is guaranteed that the same edges will be processed before pruning on every execution.

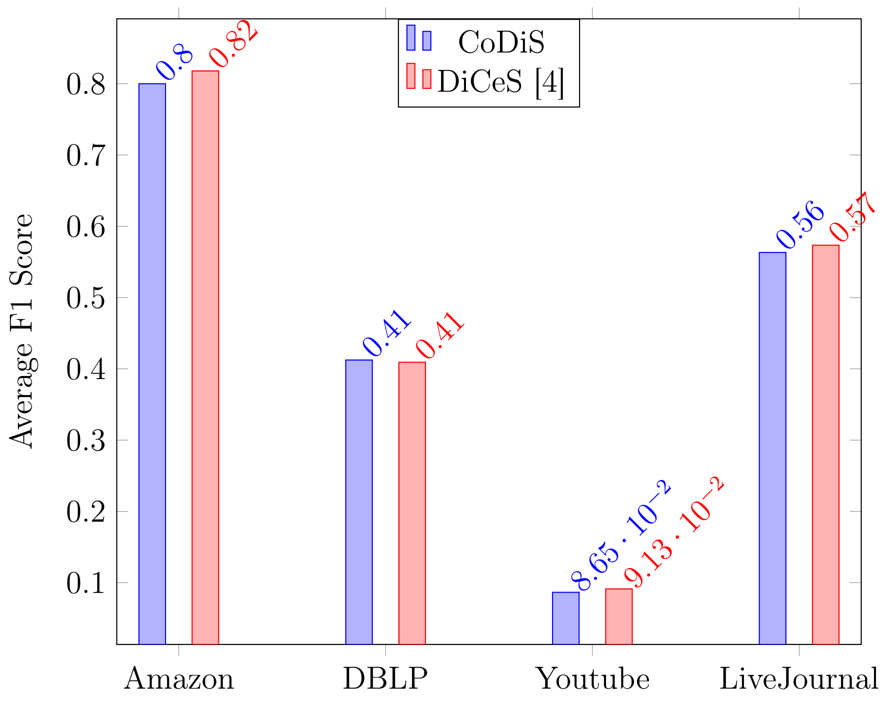


Figure 13: Comparison of average F1 scores for DiCeS [4] and CoDiS, averaged over 10 runs

5.2 Execution Time Evaluation

The main goal of our approach was not to improve the accuracy of [4], but instead to improve the running time. The main metric that we are looking at to validate this is the average time it takes to process a single edge. This metric was calculated by measuring the entire runtime of the program (in microseconds) after community initialization and then dividing that by the number of edges processed. This gives a good average time without the overhead required to time each individual edge and average them at the end.

Figure 14, as well as table 2, show the average time-per-edge of CoDiS and DiCeS at different worker node counts. Across the board, there was a significant decrease in time when running CoDiS as opposed to times of DiCeS [4]. The average decrease in processing time is 3.34 times faster than DiCeS [4]. At 2 bolts, the speedup is 3.26

Dataset	CoDiS 2 Bolts	DiCeS [4] 2 Bolts	CoDiS 3 Bolts	DiCeS [4] 3 Bolts	CoDiS 4 Bolts	DiCeS [4] 4 Bolts
Amazon	17	46	14	35	11	32
DBLP	10	47	9	42	7	38
Youtube	26	69	19	53	17	48
LiveJournal	19	57	15	41	12	38

Table 2: The time-per-edge of CoDiS and DiCeS [4] for each dataset at 2, 3, and 4 processing nodes

times greater, at 3 it is 3.17 times greater, and at 4 it is 3.58 times greater.

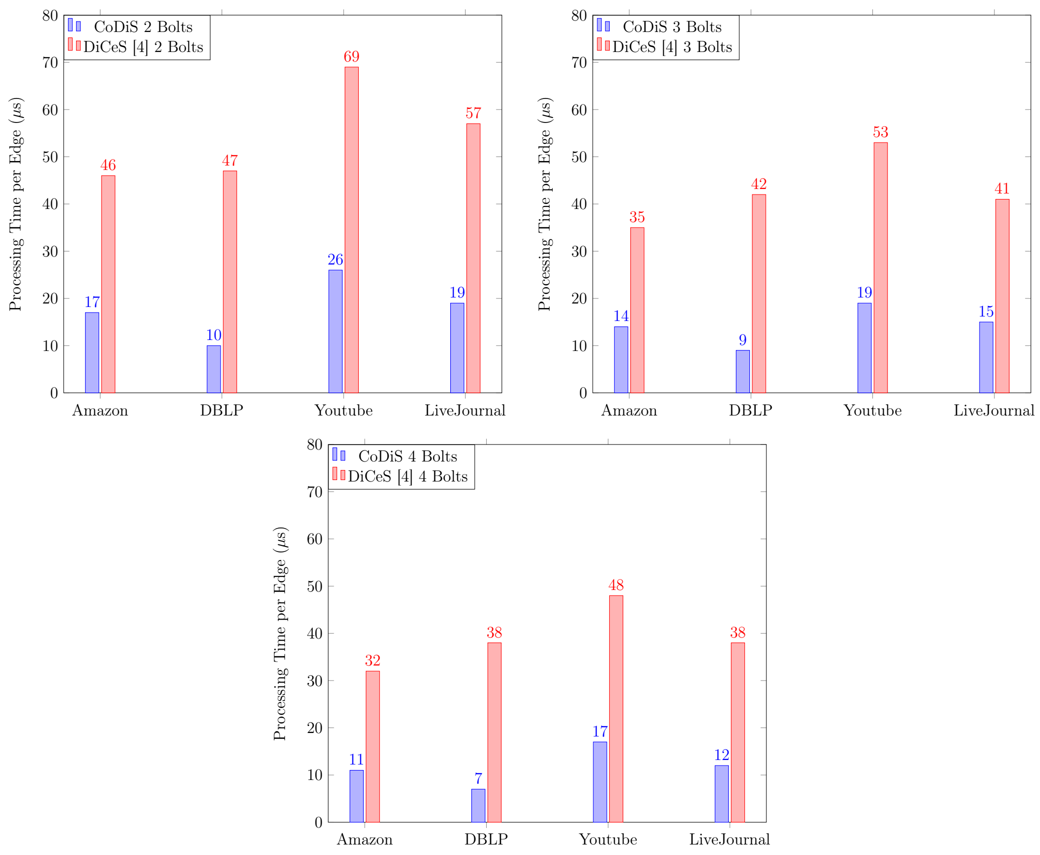


Figure 14: Comparison of average time per edge for CoDiS and DiCeS when run on our test machine with 2, 3, and 4 processing nodes, averaged over 10 runs

5.3 Space Usage Evaluation

While the goal of our approach is to improve the runtime, there is unfortunately a negative effect on the space used because of our changes. In addition to keeping track of the community degree of every node for every community, both CoDiS and DiCeS need to keep track of the total degrees as well. Because the data is no longer stored in a centralized location, each individual worker node has to maintain the total degrees count itself. This means that the space required to store the node degrees scales with the number of worker nodes being deployed.

Figure 15 shows the difference in memory usage between CoDiS and DiCeS [4] when run on the same dataset and differing numbers of Bolts. This data was collected using VisualVM to monitor the heap size and actual heap utilization of the JVM as the program ran. The average memory usage of DiCeS [4] stays relatively constant even when the number of bolts changes, whereas the memory usage of CoDiS scales almost linearly with the number of bolts.

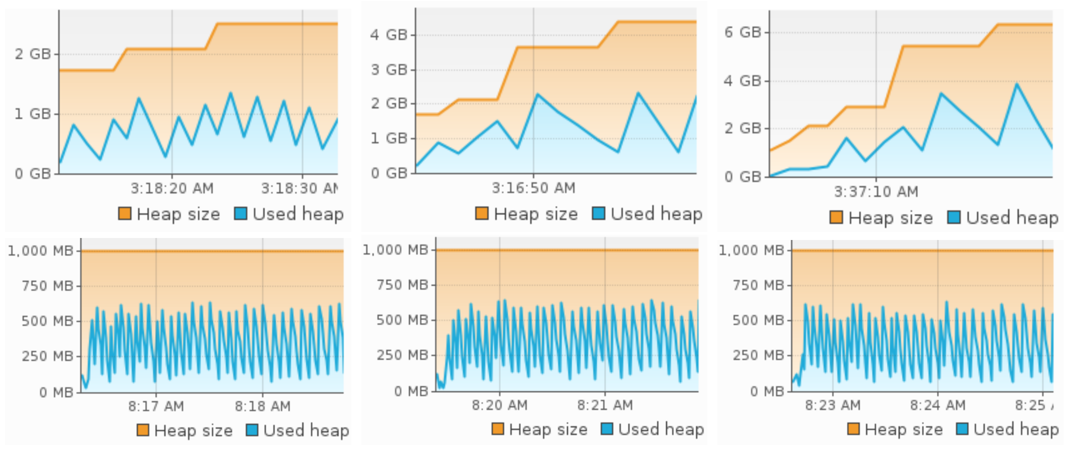


Figure 15: Top: The memory usage of CoDiS at (left to right) 2, 3, and 4 Bolts
Bottom: The memory usage of DiCeS [4] at (left to right) 2, 3, and 4 Bolts

However, we believe that these results are not actually indicative of a real-world application. While this increase in space usage is obvious when running on a single system in local mode, we believe that this discrepancy in space usage would be significantly reduced in an actual distributed-computing setup. Because of the nature of a distributed-computing deployment, the data will not be duplicated on any single machine and is instead shared among all the machines. In addition, in an actual deployment, the data of the Redis cluster is sharded between all the machines, meaning the data access time would be even longer than when run on a single machine as in testing.

5.4 Issues with Non-Shuffled Input Graphs

During our testing, we found that there was a large discrepancy between the results reported in [4] and the results we were seeing when run on our test system. Figure 16 shows the edge times that were reported by [4]. The shortest time to process an edge is when working with the amazon dataset, at 210 μs per edge. DBLP and Livejournal both took around 300 μs per edge, approximately 1.42 times longer per edge than when running on the Amazon dataset. Youtube took the longest at around 365 μs per edge, approximately 1.74 times longer than Amazon.

When running the source code for [4] on our test system, which uses a much faster CPU, we found that it only took 38 μs on average to process an edge for the Amazon dataset. If the times followed the same scaling as in [4], we would expect the time per edge of the DBLP dataset to be $38 * \frac{300}{210} = 54.28\mu$s, which is almost exactly what we saw when actually running it on our machine.

Unfortunately, that is where the similarities stop, as both the Youtube and LiveJournal datasets had *significantly* higher processing time per edge than we expected, with LiveJournal taking 4 times longer than expected and Youtube taking a whopping 31 times longer. Figure 17 shows the expected running time-per-edge of DiCeS on our machine, extrapolated from the running time of the Amazon dataset, and the results from [4]. This increase in time-per-edge was immediately apparent, the processing time did not increase as the program ran but instead was longer from the very first

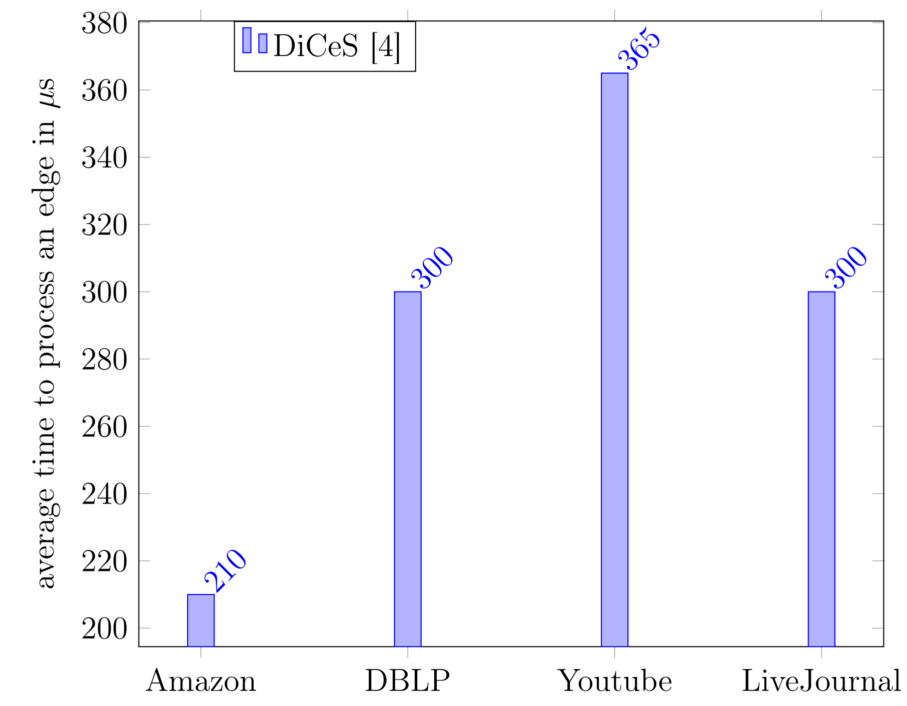


Figure 16: The edge times reported by [4] from testing on their machine using 4 processing bolts

edge.

In order to confirm that it wasn't an issue with our test machine, we ran the tests on 2 other machines (both running windows 10): a desktop system with an i7-7700k and 32 GB of RAM, and a laptop with a Ryzen 9 4900 HS and 16 GB of RAM. While the time per edge of these systems varied due to the different specifications of them all, we did in fact see a large jump when being run on the Youtube and LiveJournal data-sets.

Thankfully, with the help of Dr. Liakos, one of the authors of [4] and the individual who maintains the GitHub repository for the paper, we were able to determine the cause of the problem. Dr. Liakos informed us that not only were the seed-sets randomized between each test but the input graph file was shuffled as well. Figure 18 shows that after performing this shuffling, the runtime of DiCeS was much more

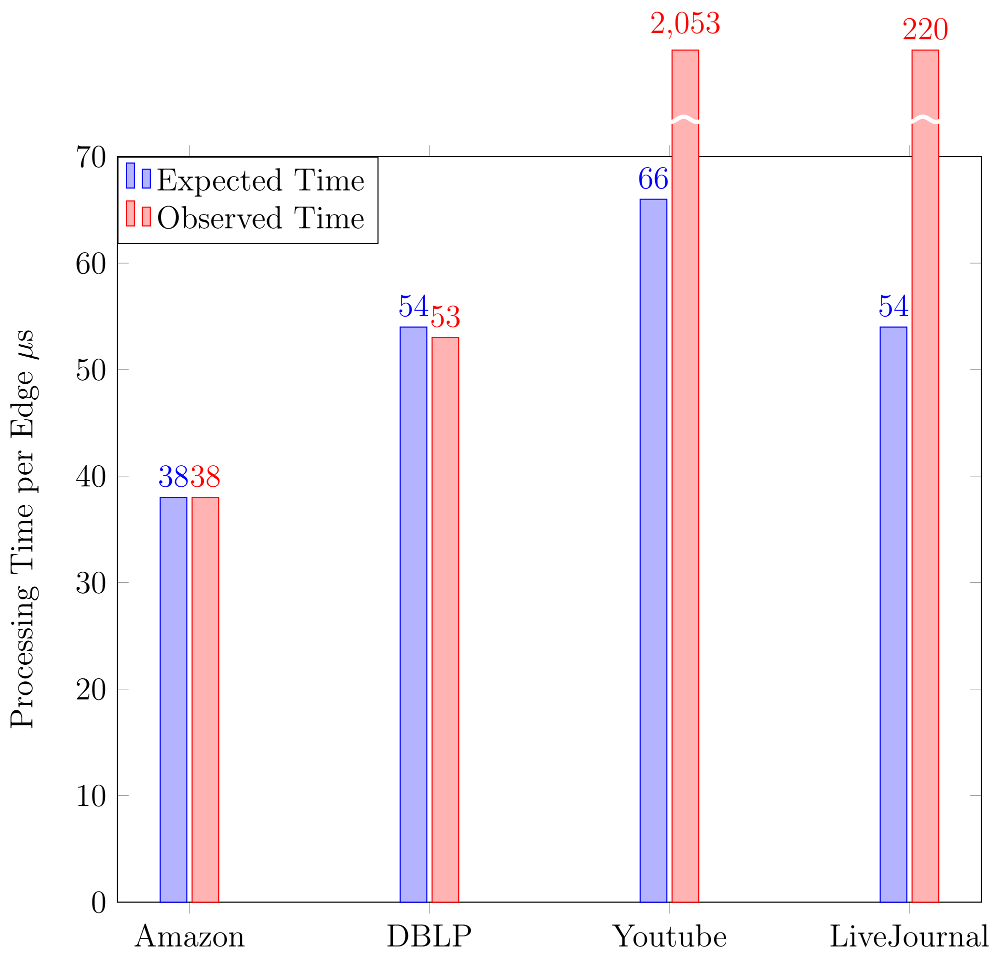


Figure 17: Comparison of the expected DiCeS [4] time per edge vs the actual time per edge on the YouTube dataset with 4 bolts

in line with what we expected to see.

The reason this occurs is that in a dense graph like Youtube, there can be hundreds of edges to just a single node. Because of this, any communities with that node in the seed set will quickly grow very large before pruning can occur, causing the processing for those communities to become exceptionally long. Additionally, because all the edges are going to many of the same communities, many worker nodes are all trying to access the same resources at once, leading to a large bottleneck. We believe that this is the reason we did not see the same exponential increase in time with an

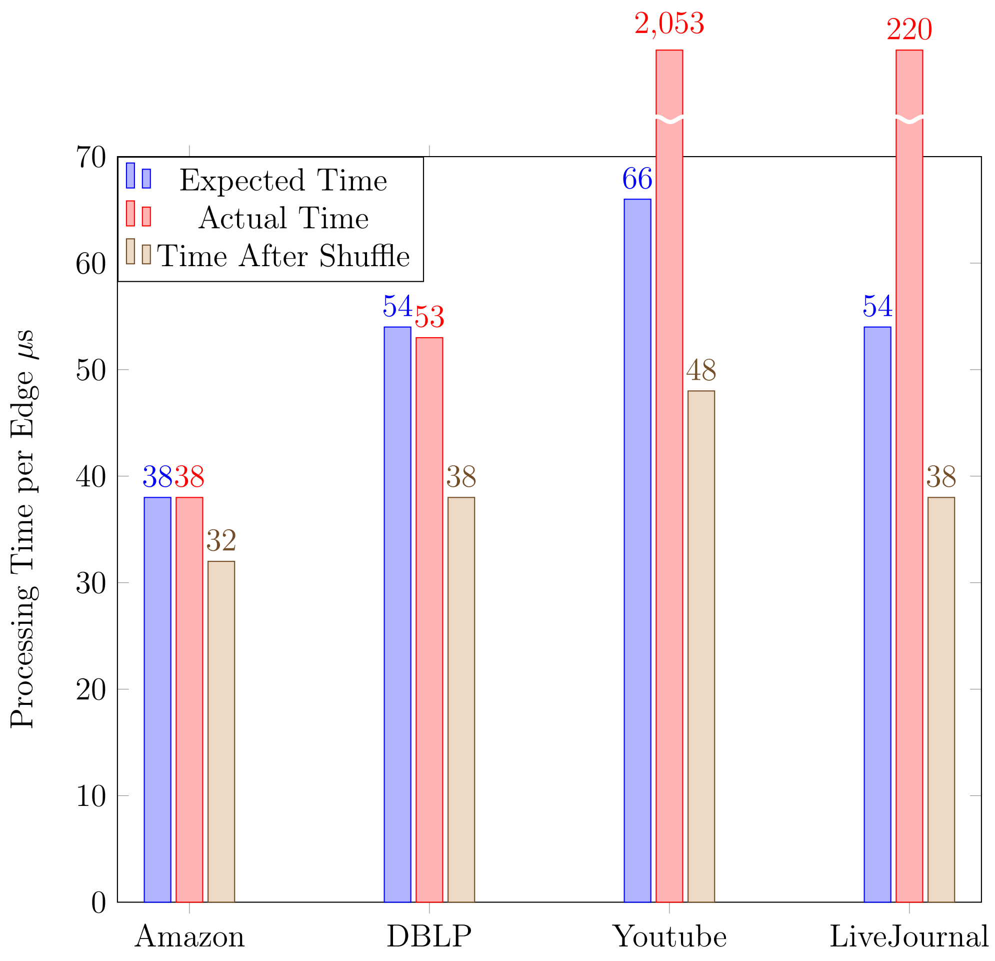


Figure 18: Comparison of DiCeS [4] running times observed after shuffling the input graph, as opposed to the expected and observed times from the non-shuffled input graph

unshuffled graph in CoDiS, as even if a worker node receives several edges that go to the same community in a row, it can quickly throw them out if those edges don't belong to any of its communities. Even if they do, the worker node doesn't have to wait for other workers to access any resources, and can immediately process the node.

This situation is not unlikely to occur naturally. Consider a web crawler that emits as edges links between webpages. If this crawler comes across a page with a large number of links to other pages and emits all those links before crawling to any of them, it would be the same scenario as reading an unshuffled graph, as we would receive several edges in a row that all share at least one node in common.

CHAPTER 6

Results and Conclusion

In this paper, we propose a method of distributing the work of the CoEuS algorithm that is an alternative to the method proposed by [4]. This new method takes inspiration from the subgraph approach and takes advantage of the unique way [5] approaches community detection to process the communities themselves in parallel instead of individual edges. By doing this, we hope to gain significant time savings by removing the need to distribute data among systems and concurrency protection, while having little to no impact on the accuracy of the found communities.

In addition, we discovered a previously unknown downside to [4]: a significant loss in processing speed when the input is a long run of edges that all share the same node. This situation is not unlikely to be encountered in a real deployment, so we believe the fact that our implementation is not vulnerable to this event is another improvement over the original approach.

Chapter 5 shows that we successfully achieved this goal. Our proposed implementation had little to no change in F1 score, with the difference in average F1 score being smaller than the largest seen variation in F1 score between runs of [4]. Based on these results, we believe that our implementation is a significant improvement to [4], and can even be improved further.

6.1 Future Work

While the work we have done to implement our changes has already shown significant improvement, there is still more we would like to do if we get a chance to revisit this topic in the future

6.1.1 Custom `SortedMap`

In [4], all data is stored in a `Redis` cluster. The community nodes specifically are stored in what Redis calls a `SortedSet`, which is essentially a `String` to `Double` map that is sorted on the value of the `Double`. This allows Redis to quickly remove the last X values from the `SortedSet`. As removing Redis was one of the major goals of our implementation, we were unable to use this dataset, and instead used a Java `HashMap`. Because of this, whenever we want to prune a community, we need to get the map pairs as a list, sort that list, get only the first X values that we want to keep, create a new `HashMap` from those pairs, and then replace the current `HashMap` with the new one. This adds a lot of time and space complexity to our approach, and in the future, we would like to create a custom datatype that can be kept sorted and therefore easy to prune quickly while not losing the fast access time of a `HashMap`.

6.1.2 Parallelize Pruning

One of the advantages of making the community data available to all processing nodes is that it is possible to move the pruning of the communities to its own processing node. By removing Redis from our implementation, we now have to do the pruning sequentially with the edge processing, meaning we have to pause edge processing any time we want to prune. In the future, we would like to look into ways to possibly move the pruning process back to being done in parallel to the edge processing. However, this might become unnecessary if the pruning process could be accelerated using a custom datatype.

LIST OF REFERENCES

[1] M. Girvan and M. E. J. Newman, "Community structure in social and biological networks," *Proceedings of the National Academy of Sciences*, vol. 99, no. 12, p. 7821–7826, Jun 2002.

[2] S. Gregory, "An algorithm to find overlapping community structure in networks," in *Knowledge Discovery in Databases: PKDD 2007*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 91–102.

[3] H. Sun, W. Jie, J. Loo, L. Wang, S. Ma, G. Han, Z. Wang, and W. Xing, "A parallel self-organizing overlapping community detection algorithm based on swarm intelligence for large scale complex networks," *Future Gener. Comput. Syst.*, vol. 89, pp. 265–285, 2018.

[4] P. Liakos, K. Papakonstantinopoulou, A. Ntoulas, and A. Delis, "DiCeS: Detecting communities in network streams over the cloud," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, 2019.

[5] P. Liakos, A. Ntoulas, and A. Delis, "COEUS: Community detection via seed-set expansion on graph streams," in *2017 IEEE International Conference on Big Data (Big Data)*. IEEE, 2017.

[6] M. E. J. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Phys. Rev. E Stat. Nonlin. Soft Matter Phys.*, vol. 69, no. 2 Pt 2, p. 026113, 2004.

[7] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *J. Stat. Mech.*, vol. 2008, no. 10, p. P10008, 2008.

[8] S. Gregory, "A fast algorithm to find overlapping communities in networks," in *Machine Learning and Knowledge Discovery in Databases*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 408–423.

[9] S. Gregory, "Finding overlapping communities in networks by label propagation," *New J. Phys.*, vol. 12, no. 10, p. 103018, 2010.

[10] J. Xie, B. K. Szymanski, and X. Liu, "SLPA: Uncovering overlapping communities in social networks via a speaker-listener interaction dynamic process," in *2011 IEEE 11th International Conference on Data Mining Workshops*. IEEE, 2011.

[11] K. Kuzmin, S. Y. Shah, and B. K. Szymanski, “Parallel overlapping community detection with SLPA,” in *2013 International Conference on Social Computing*. IEEE, 2013.

[12] Y. Zhang, D. Yin, B. Wu, F. Long, Y. Cui, and X. Bian, “PLinkSHRINK: a parallel overlapping community detection algorithm with Link-Graph for large networks,” *Soc. Netw. Anal. Min.*, vol. 9, no. 1, 2019.