K-edge Connected Components in Large Graphs: An Empirical Analysis

by

Hanieh Sadri

B.Sc. (Applied Mathematic), Shahid Beheshti University, 2020

A Thesis Submitted in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

K-edge Connected Components in Large Graphs: An Empirical Analysis

by

Hanieh Sadri

B.Sc. (Applied Mathematic), Shahid Beheshti University, 2020

Supervisory Committee

Dr. A. Thomo, Supervisor

(Department of Computer Science)

Dr. V. Srinivasan, Co-supervisor

(Department of Computer Science)

ABSTRACT

Graphs play a pivotal role in representing complex relationships across various domains, such as social networks and bioinformatics. Key to many applications is the identification of communities or clusters within these graphs, with k-edge-connected components emerging as an important method for finding well-connected communities. Although there exist other techniques such k-plexes, k-cores, and k-trusses, they are known to have some limitations.

This study delves into four existing algorithms designed for computing maximal k-edge-connected subgraphs. We conduct a thorough study of these algorithms to understand the strengths and weaknesses of each algorithm in detail and propose algorithmic refinements to optimize their performance. We provide a careful implementation of each of these algorithms, using which we analyze and compare their performance on graphs of varying sizes. Our work is the first to provide such a direct experimental comparison of these four methods. Finally, we also address an incorrect claim made in the literature about one of these algorithms.

Keywords: Graph, Community Detection, k-Edge-Connected Components, Clustering, Well-Connected Communities

Table of Contents

List of Tables

List of Figures

ACKNOWLEDGEMENTS

I would like to extend my deepest gratitude to:

Dr. Thomo, for not only giving me the incredible opportunity to pursue this degree but also for being a constant source of inspiration and support. Your mentorship has been invaluable in shaping my academic journey and your encouragement has been a guiding light through challenging times.

Dr. Srinivasan, for generously sharing your vast knowledge and expertise. Your mentoring has profoundly impacted my understanding and approach to my field of study. Your patience and thoughtful guidance have been instrumental in my growth both as a scholar and as an individual.

My family, for your unwavering support throughout this journey. Your belief in me has been the foundation of my strength and perseverance. Your sacrifices and encouragement have made all the difference, and this achievement is as much yours as it is mine.

Chapter 1

Introduction

Graphs have become increasingly important in today's world due to their ability to capture complex relationships and provide valuable insights [30]. In practical scenarios, we often encounter various data and their relationships which can be effectively depicted using graphs. For instance, they are used in areas like social networks [27], web searches [31], biochemistry [18], biology [3], and road network mapping [8]. Given their widespread use and significance, a lot of research is being conducted to analyse graph data [23].

Identifying communities within graphs, which are essentially clusters of densely connected vertices, is a vital concept due to its wide-ranging applications [13]. In social networking platforms, community detection can be leveraged for friend recommendations, targeted social campaigns, and advertising [42]. Within protein-protein interaction networks, community detection can be applied to recognizing proteins with similar functionality [36]. Meanwhile, in a web-link-based graph, a community might represent a set of web pages sharing substantial commonality, aiding in finding similarities among them [4].

One prevalent method for identifying such clusters and communities is by comput-

ing k-edge-connected components (cf. [43, 1, 6, 37, 7]). These components are induced maximal subgraphs that remain connected after removing $k-1$ edges. While there are alternative notions of graph density, such as k-core (cf. [34, 22, 11]) or k-truss (cf. [39, 41, 10]) components, k-edge-connected components often give well-connected communities that k-core and k-truss fail to discover (cf. [1]). In detail, a k-core component is a maximally connected subgraph such that all vertices in it have a degree at least k. There is a limitation to considering k-core components as clusters because sometimes a k-core component may not be able to separate well-defined communities. On the other hand, a k-truss component is a maximally connected subgraph in which every edge is contained in at least $k-2$ triangles. Using k-truss for community detection can also be limiting, especially k-trusses may fail to identify cohesive groups in the absence of triangles. In such cases, k-edge-connected components emerge as a more viable solution.

Our work aims to provide a detailed exploration of four main algorithms for computing k-edge-connected components. Our approach implements each of these algorithms, comparing their performance, and suggesting optimization strategies where applicable.

The first algorithm we consider, presented in [6], is based on graph decomposition. Its main idea is to decompose the graph until all the remaining connected subgraphs are k-edge connected.

Another algorithm that we explore, introduced in [1], is based on contracting random edges. The idea behind this method is repeatedly finding cuts with sizes less than k and dividing the graph along these cuts. If we reach the point that each connected component has no cut with a size less than k, then they are k-edge-connected.

Subsequently, we investigate another algorithm called the early merge and split

method, presented in [37], which proposes an improvement to the decomposition algorithm of [6]. Its core idea is to combine vertices that meet the k-connectivity requirement.

Lastly, we study a theoretical algorithm, given in [7], that computes k-edge connected subgraphs by computing specialized cuts. This algorithm is mainly in the theoretical realm and has resisted implementation until now, which we present in this work.

1.1 Contributions

The main contributions of this paper are summarized as follows:

1. We undertake a thorough implementation and experimental study of the four aforementioned methods for computing k-edge-connected components. Our comprehensive analysis assesses each method's strengths, limitations, and applicability. To ensure a fair comparison, we implemented each method in the same programming language and evaluated them under identical environments and setups, utilizing a range of datasets from small to large.

2. An important contribution of our study is our capability to extract the unique features of each method, equipping us to optimize each one effectively. Additionally, we carefully engineer certain algorithms, leading to marked improvements in their scalability.

3. Our study features a direct comparison of the algorithms from [6] and [1]. Since both were published around the same time, a direct comparison had not been previously conducted, resulting in a gap in comparative analysis. Our work fills this void, providing valuable insights into their relative performance, strengths, and weaknesses.

4. We highlight and correct inaccurate claims made by [37]. Contrary to their claims, we demonstrate that their proposed improvements do not perform as effectively in practice.

5. We provide the first implementation of a theoretical algorithm presented in [7] to identify k-edge-connected components. This algorithm, due to its abstract nature, had not been previously implemented, even by its original authors. Our implementation sheds light on its practical efficiency and applicability.

1.2 Organization

This thesis is organized as follows:

- **Chapter 2: Preliminaries** - This chapter introduces the essential notations and definitions related to graphs employed throughout this work. It also provides a comprehensive overview of existing algorithms and methods for identifying and analyzing communities within graphs.

- **Chapter 3 Algorithms and Implementations** details the four algorithms we explored.

- **Chapter 4 Experiments** presents our experimental settings and results.

- **Chapter 6 Conclusions and Future Work** concludes the thesis and offers insights on how to improve the algorithm further.

Chapter 2

Preliminaries

This chapter establishes the fundamental definitions and background concepts utilized in the thesis. It also delves into a range of algorithms specifically associated with the detection of communities in network graphs.

2.1 Definitions

In this section, we begin by introducing some terminology and definitions. Our work deals with undirected and unweighted graphs.

Definition 1. *An* **undirected graph** G *is denoted as* $G = (V, E)$*, where* V *is the set of vertices and* E *is the set of undirected edges. Furthermore, we denote by* n *and* m *the sizes of* V *and* E*.*

The notions of connectivity of a graph and the degree of a vertex are central to our work.

Definition 2. *An undirected graph* G *is* **connected** *if for any two vertices* $u, v \in V$*, there is a sequence of edges* $(u, v_1), (v_1, v_2), \ldots, (v_k, v)$ *between* u *and* v *in* G*.*

Definition 3. *For a vertex $v \in V$ in $G = (V, E)$, the* **degree** *of v, denoted as $deg(v)$, is defined as the number of edges incident to v.*

Next, we describe the notions of induced subgraphs and cuts in graphs needed to study k-connectedness.

Definition 4. *For a vertex subset $V' \subseteq V$, the subgraph $G[V']$* **induced** *by V' is the subgraph of G with vertex set V' and the edge set $E' \subseteq E$ that only includes the edges from G connecting vertices both in V', i.e. $E' = \{(u, v) \in E \mid u, v \in V'\}$.*

Definition 5. *A* **cut** *in a graph $G = (V, E)$ partitions its vertices into two disjoint subsets S and T. The cut set, denoted $C(S, T)$, consists of all edges with one endpoint in S and the other in T, whose removal disconnects G. We refer to $|C(S, T)|$ as the size of the cut.*

We now formally define the problem we study.

Definition 6. *A graph G is described as k-**edge-connected** if it continues to be connected even when any $k - 1$ edges are removed. In other words, G is k-**edge-connected** if the minimum size of of a cut in G is at least k. A k-edge-connected component of a graph G is an induced subgraph H of G that remains connected even after the removal of any $k - 1$ edges, and there is no larger subgraph in G with this property that contains H.*

For illustration, Fig 2.1 displays a 2-edge connected graph; even after the removal of one edge, it stays connected. Within this, there are three 3-edge connected subgraphs $G1$, $G2$, and $G3$, each of which remains connected even after two edges are removed, as highlighted by the dotted circles. It is important to note that these 3-edge connected components are unique; meaning that whenever algorithms are applied for finding such components, the same results will be consistently obtained.

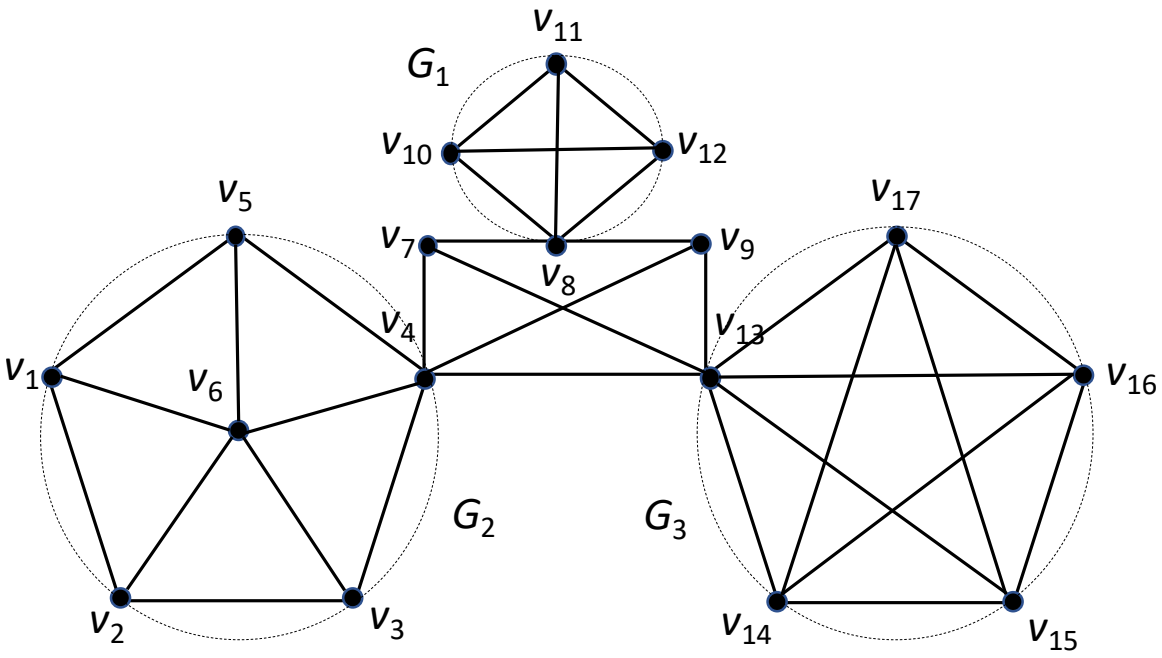


Figure 2.1: The 3-edge connected components of a graph

2.2 Related Work

2.2.1 k-connectivity

Given a graph G. A k-edge connected component within G is defined as a maximal induced subgraph that retains connectivity despite the removal of any $k-1$ edges. Various methodologies have been developed to identify these components. Decomposition-based algorithms aim to identify k-edge-connected components by iteratively decomposing the graph into connected components until each is a k-edge connected component [6, 37, 1]. On the other hand, cut-based algorithms iteratively cut a graph with connectivity less than k into two parts, continuing this process until the connectivity of all resulting subgraphs is at least k [7, 43].

In the realm of graph theory, there is also an alternate definition for k-edge-connected components, which we are not examining in this study. According to this other interpretation, a k-edge-connected component is recognized as a maximal

group V_i of vertices within graph G, where each vertex pair is k-connected in the entire graph G. However, this does not guarantee the same connectivity within the induced subgraph created by V_i, which could be disconnected. Identifying all such maximal vertex groups presents a unique challenge, different from identifying the k-edge connected components of G, which is our research focus. Several studies have explored this alternative concept [14, 28, 29, 38].

2.2.2 All-k-edge-connected-components

The challenge of identifying all k-edge-connected components involves determining the k-edge-connected components of graph G for all possible values of k. The methodology suggested in [5] revolves around the construction of a hierarchy tree for G, which effectively finds the k-edge-connected components for all possible k values. The work in [40] identifies all k-edge-connected components, using a definition that targets the discovery of maximal vertex subsets wherein each pair of vertices is k-edge connected within the graph G. Finding all k-edge-connected components is beyond the scope of this study, and we concentrate on identifying k-edge-connected components for a specific k.

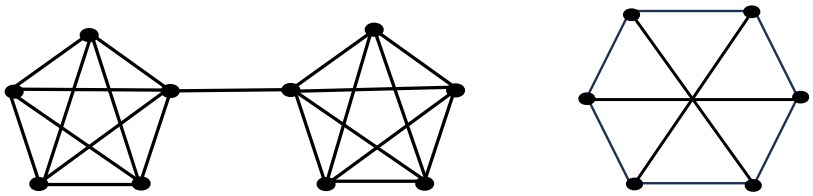
2.2.3 Other notions of dense subgraphs

Here, we discuss other algorithms that have been leveraged to find sets of closely related vertices in a graph.

Degree based algorithms: A clique, also called complete subgraph, is a subset of vertices within a graph, where each vertex is adjacent to every other vertex within the subset. Since the condition of being a clique is too strict, using cliques we often miss important clusters. A k-plex is a generalization of a clique within a graph, allowing for a more flexible and inclusive definition of community structures. Unlike

a clique, where each node is connected to every other node, a k-plex relaxes this requirement to allow for a specified number of non-adjacencies. In a k-plex, each node is required to have an edge to at least $n - k$ other nodes within the k-plex, where n is the total number of nodes in the k-plex. That is, the degree of every vertex in the induced subgraph is at least $n - k$ [2]. Identifying the maximal cliques and k-plexes in graphs is NP-hard, presenting a significant computational challenge for exact solutions.

A k-core is a maximal subgraph within a graph where every vertex has a degree of at least k. In the context of community detection and clustering, this method might sometimes not produce the desired well-connected clusters. More specifically, a k-core component can be divided into k-edge-connected components. These components can be interpreted as a refinement of k-core subgraphs. For instance, in figure 2.2.a, the graph is a 4-core by definition. Nevertheless, it can be separated into two 4-edge-connected components.



(a) (b)

Figure 2.2: Challenges in cluster identification by conventional algorithms: (a) Example of an ill-connected cluster not identified by algorithms like k-core and r-clique; (b) shows a well-connected cluster that k-truss fails to recognize.

Distance based algorithms: An r-clique is defined as a maximal subgraph where the largest distance between any two nodes does not exceed r [32]. An r-clan is simply an r-clique with the additional condition that its diameter does not exceed r. An r-club is a maximal subgraph with a diameter limited to r. Finding all these structures is known to be NP-Hard. Also since these concepts do not distinguish

subgraphs with both low diameter and weak connectivity, applying these algorithms may result in ill-connected clusters.

Triangle based algorithms: For a graph G, a subgraph G' is described as a k-truss if every edge e within G' is part of at least $k-2$ triangles. Furthermore, G' must be the maximal subgraph with this property [39]. Utilizing k-truss for community detection can be restrictive, especially for bipartite networks where there are few or no triangles. In such scenarios, k-edge-connected components present a more practical alternative. For example in Figure 2.2.b, the graph is 2-edge connected. However, it is impossible to identify any truss due to the absence of triangles in the graph. Similar limitations hold for other closely related notions [12].

Chapter 3

Algorithms and Implementations

In this chapter, we present a thorough analysis of algorithms used to identify k-edge-connected components in graphs. Each algorithm is described in terms of its foundational principle, accompanying pseudocode, and its asymptotic runtime. Moreover, we shed light on specific enhancements implemented to boost the performance of certain algorithms. Among the studied methods, the graph decomposition algorithm emerged as the most effective for determining k-edge-connected components. On the other hand, the random contraction method, despite having a similar runtime complexity, exhibited markedly slower performance, especially with larger graph sizes. Recognizing its constraints, we refined its data structures, achieving a version that is approximately five times faster in empirical tests and better suited for larger networks.

We further examine the early merge and Split algorithm, which is similar to the graph decomposition algorithm. Nonetheless, its efficacy is overshadowed by the graph decomposition and random contraction methods. Lastly, we explore the local cut detection algorithm, which is designed for identifying k-edge-connected components for directed graphs. Although the theoretical time complexity of this algorithm

is close to linear, our findings suggest its scalability is constrained, particularly for medium to large datasets.

3.1 Decompose-LMS Algorithm

An important algorithm called Decompose-LMS for identifying k-edge-connected components was introduced by Chang et. al in [6]. The core idea of the algorithm involves iteratively decomposing graph G. During each iteration, the algorithm identifies subgraphs that are not k-edge-connected and further decomposes them. This process continues until all resulting subgraphs are k-edge-connected. The outcome is a list of k-edge-connected components. High-level pseudocode for this algorithm is provided in Algorithm 1.

Algorithm 1 Computing k-Edge-Connected Components

```
1: Input: A graph G = (V, E) and an integer k.
2: Output: k-Edge-connected components of G.
3: Initialize a queue Q_g with the graph G as its sole member.
4: for each subgraph g in Q_g do
5:     φ_k(g) ← Decompose(g, k);
6:     if φ_k(g) consists solely of one subgraph then
7:         Output φ_k(g) as a k-edge-connected component;
8:     else
9:         Enqueue all subgraphs of φ_k(g) into Q_g;
```

A key part of the decomposition involves repeatedly applying merge and split operations on what is called the partitioned graph. The idea of a partitioned graph PG is taking a graph G, consisting of vertices V and edges E, and appending additional information to each vertex from a domain D. This is done to track modifications in the graph after vertex merging. In an ordinary graph, vertices do not have extra associated information, thus $D(u) = \{v\}$ for all v in V. However, upon applying a merge operation on vertices v_i and v_j these vertices combine into a single super-vertex, u.

Specifically, u is added to PG such that $D(u) = D(v_i) \cup D(v_j)$. Then, edges (u, x) are added to PG if x belongs to the neighbor set of either v_i or v_j. If a vertex x is adjacent to both v_i and v_j, in the resulting partition graph a parallel edge from u to x is created. Following this, vertices v_i, v_j, and their linked edges are removed from PG.

To simplify the discussion, we blur the distinction between a graph and a partitioned graph. Also, when we do not explicitly show the associated set of a vertex, we assume that the set is the singleton set containing the vertex itself.

Example 1. *Consider the graph in Figure 3.1.a. If we create a partition graph of G, since there is no information at the beginning for the vertices, we have $D(v_0) = v_0$, $D(v_1) = v_1$, $D(v_2) = v_2$, and $D(v_3) = v_3$. Figure 3.1.b shows the partition graph after merging vertices v_0 and v_1. As can be seen, the vertex u is added to the graph such that $D(u) = D(v_0) \cup D(v_1) = \{v_0, v_1\}$.*

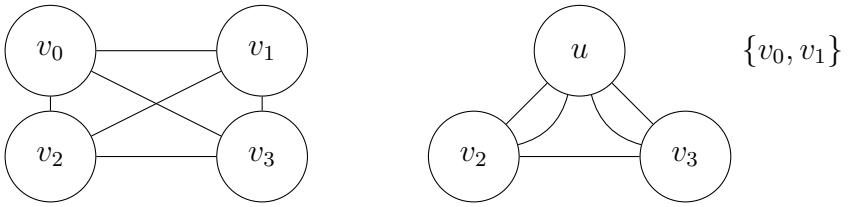


Figure 3.1: Partition graph before and after applying a merge operation

An important part of the decompose algorithm is creating several connected subgraphs by removing the edges in a cut of G with value less than k. This relies on the *Maximum Adjacency Search (MAS)* procedure to compute the minimum cut between a pair of vertices.

MAS organizes all vertices in the graph G into a list L. Assume the last vertex in

this list is t, and the vertex immediately preceding it is s. In this configuration, the edges adjacent to t in G constitute the minimum $s-t$ cut.

The construction of the list L initiates by randomly selecting a vertex from V and adding it to L. As long as there are remaining vertices not yet in L, a vertex $u \notin L$ is selected. This vertex u is the one with the maximum number of connections to L, mathematically represented as $u = \arg\max_{v \in V \setminus L} w(L, v)$, where $w(L, v)$ denotes the number of edges connecting v with vertices in L. The selected vertex u is then appended to the end of L.

To optimize the identification of the most tightly connected vertex in *MAS*, a specialized data structure is introduced.

Let $key(v)$ represent the key of vertex v during the execution of *MAS*, where $key(v) = w(L, v)$, indicating the number of edges between v and vertices in L.

In this data structure, doubly linked lists are employed alongside a head array. The head array specifically holds the first vertex associated with each key value x; to clarify, Head$[x] = v$, where $v \in V$, signifies that v is the head vertex in the doubly linked list with a key value of x. Within this linked list arrangement, three arrays—each with a size of $|V|$—maintain the key, and "next", and "previous" indexes (pointers) for each vertex in the list.

Example 2. *Refer to Figure3.2 for an illustrative example that visualizes the data structure. From this figure, we observe that* $Head(x) = v_i$*. To identify the subsequent vertex with a key equal to* x*, we can use* $next(v_i)$*. Here we have that* $next(v_i) = v_j$ *and* v_j *also has a key of* x*. Furthermore, we see* $previous(v_j) = v_i$*.*

In addition to the head array and doubly-linked list, we also maintain a value p_0, which represents the current maximum key value among all vertices in the data structure. This value is initialized to 0 and is updated whenever a new vertex is inserted into the data structure.

To update the key of vertex v from x to y, we first remove v from the doubly-linked list represented by Head(x), and then insert v into the doubly-linked list Head(y). Additionally, *max-key* is updated to y if $y > p_0$.

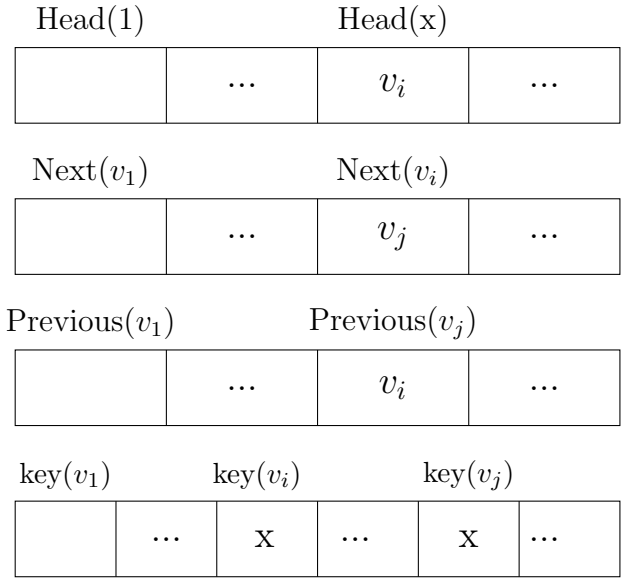


Figure 3.2: Data structure

To extract the next vertex with the largest key value, we first decrement p_0 until Head(p_0) is not null. We then report the first vertex pointed to by Head(p_0) and remove that vertex from the doubly-linked list.

Algorithm 2 integrates the aforementioned ideas into an efficient method for implementing decomposition. It employs linear data structures and incorporates early merge and list sharing optimizations explained in the following.

During a single iteration of *MAS-LMS*, it is possible to merge multiple pairs of vertices, provided that each pair is guaranteed to be k-connected in the input graph. This strategy is known as Early Merge. If *MAS-LMS* identifies a minimum cut smaller than k, the minimum cut of the graph produced by the subsequent split operation can be directly obtained from the existing list L. This eliminates the need to execute *MAS-LMS* on the newly formed graph. The capability to reuse the list L across

multiple instances of *MAS* following split operations is termed List Sharing.

The time complexity of the *Decompose-LMS* is given by $O(l \cdot |E|)$, where l denotes the number of repetitions of the *MAS-LMS* procedure. The overall time complexity of Algorithm 1 is $O(h \cdot l \cdot |E|)$, where h is a parameter representing the number of times *Decompose-LMS* is called within Algorithm 1. As pointed out by [6], h is typically a small integer for real-world graphs.

Algorithm 2 Decompose-LMS

Input: A graph $G = (V, E)$ and an integer k.
Output: Subgraphs of G if $\delta(G) < k$, and G otherwise.
Construct the corresponding partition graph PG_0 of G
$i \leftarrow 0$
while The edge set of PG_i is non-empty **do**
    $PG_{i+1} \leftarrow$ MAS-LMS(PG_i, k)
    $i \leftarrow i + 1$
Return $\phi_k(PG_i)$

Function MAS-LMS(G, k)
$L \leftarrow \{$an arbitrary vertex u of $V\}$
Initialize the data structure
while $L \neq V$ **do**
    $u \leftarrow$ extract-max
    Add u to L and remove u from the data structure
    Initialize a queue Q with u
    while $Q \neq \emptyset$ **do**
        $v \leftarrow Q$.pop()
        for each $(v, s) \in E$ with $s < L$ **do**
            if the key of s increases to pass k **then**
                Add s to Q, remove s from the data structure
            else
                Update-key for s
        if $u \neq v$ **then**
            Merge u and v
while $|L| > 1$ and the value of the cut implied by the last two vertices in L is less than k **do**
    Split the cut
    Remove the last vertex from L

Example 3. *Consider a graph shown in Figure 1 with $k = 3$. In the Decompose_LMS algorithm, we start by creating a partition graph, which initially is the graph itself. We then run MAS_LMS on this partition graph. Within MAS_LMS, we arbitrarily introduce vertex v_8 into the data structure with a key set to 0. We then extract v_8 and place it in L. Iterating over v_8's neighbors, we increment their keys by 1, updating the keys of neighboring vertices v_7, v_9, v_{10},v_{11} and, v_{12} which are adjacent to v_8. As none of the keys for v_8's neighbors exceed 3, nothing is added to Q.*

Following this, we retrieve the vertex with the highest key from our data structure; at this point, the highest key is 1. Assuming $head[1] = v_7$, v_7 is extracted from the data structure and then inserted into L. The keys of its neighbors, v_4 and $v_1 3$, are then updated. The key of the other neighbor which is v_8 is not updated since this vertex is already in L.

The subsequent step extracts the vertex with the highest key from the data structure, which remains 1. Assuming we extract v_9 for this key, we insert it into L. Iterating over v_9's neighbors increases the keys for v_{13} and v_4 to 2.

Thereafter, the highest key in the data structure becomes 2. Assuming v_{13} is extracted, and upon iterating over its neighbors, v_4's key increases to 3, prompting its inclusion in Q.

After completing the iteration for all neighbors of v_{13}, vertices v_{13} and v_4 are merged into a super vertex $\{v_{13}, v_4\}$. This methodology—extracting the vertex with the highest key, adding vertices to L, and merging vertices if the connectivity between them exceeds k—continues until every vertex in the partition graph becomes a part of L. Ultimately, the order in L is: v_8, v_7, v_9, $\{v_4, v_{13}\}$,v_6, v_3, v_2 , $\{v_1, v_5\}$, v_{17}, $\{v_{14}, v_{15}, v_{16}\}$, v_{10}, $\{v_{11}, v_{12}\}$ where the sets in curly braces denote super vertices.

In the subsequent phase, a while loop commences and continues until L becomes empty. As the cut between the last two vertices in L equals 3, we do not split the cut

and remove the last vertex from L. Assuming this process continues until $L = [v_8, v_7]$, the cut size between v_8 and v_7 is 2, prompting us to split the cut edges (v_8, v_7) and (v_8, v_9), and remove v_7 from L.

Subsequent MAS_LMS iterations may be required if the partition graph still contains edges. The connected subgraphs resulting from the first Decompose-LMS process are G_1 and $G_2 \cup G_3$. If more than one connected subgraph emerges, Decompose-LMS is reapplied to these subgraphs.

In conclusion, the final subgraphs G_1, G_2, and G_3 are 3-edge-connected subgraphs of the original graph G shown in Figure 1.

3.2 Random Contraction Algorithm

The concept of finding k-edge-connected components through random contractions was introduced by Akiba et. al. in [1]. The proposed algorithm is structured around a unique application of random contraction. This methodology has historically been a theoretical tool for addressing cut problems, as noted in [21]. The random contraction method of [1] involves selecting edges at random and contracting them until no edges are left in the graph. Contracting an edge involves the removal of the edge and the subsequent merging of its two endpoints.

In this algorithm, the graph is represented by a dictionary of dictionaries. For every vertex v there's an associated dictionary, h_v; the keys are neighboring vertices, while the values stand for the edge weights.

When an edge (u, v) undergoes contraction, the algorithm merges the dictionaries h_u and h_v. To ensure that this merge is carried out efficiently, the edges from the smaller dictionary are always inserted into the larger one. So, supposing that h_u is smaller than h_v, the transfer process would involve moving edges from h_u to h_v.

During this transfer, a vertex x from h_u to h_v, if h_v doesn't already contain the edge (v, x), then the algorithm simply adds the edge (v, x) to h_v. But, if h_v contains the edge (v, x), the weight of (v, x) in h_v is increased by the weight of (u, x) found in h_u. Additionally, we add v to h_x, with a weight same as the edge (v, x).

The next step involves removing u from the overall dictionary. This necessitates navigating through all of u's neighbors, as specified in h_u, and systematically excluding u from their neighbor lists. Once this step is completed, h_u can be safely removed from the graph.

Algorithm 3 for finding connected-subgraph of G by random contraction starts with generating a copy of Graph G, denoted as G'. While G' still contains edges, it randomly selects an edge for contraction. After this contraction, the degree of a vertex, say, u, might decrease and become less than k. If this happens, the subgraph formed by the vertices merged with u is added to the output. Also, all the edges connected to u are removed, and u is excluded from G'. This entire set of actions constitutes one iteration.

The iterations are then repeated for each subgraph induced by a connected component from the preceding iteration, and this is done for a predefined number of times in Algorithm 4. The precise number of iterations required depends on the graph and is determined by checking that no new output is created in some iteration. Akiba et al. show that the total time complexity of Algorithm 4 is $O(|E| \log(n))$.

One crucial observation is that while each iteration yields connected components, they may not always be k-edge-connected. A component that remains unbroken in a subsequent iteration is indeed k-edge-connected.

To streamline the process, [1] introduces a method called *forced contraction*. The underlying principle is simple: if an edge between two vertices, u and v, carries a weight of k or more, it is beneficial to contract these vertices immediately.

Algorithm 3 Basic Iteration

```
1: procedure ContractAndCut(G, k)
2:     G' ← G
3:     while G' is not empty do
4:         if exists u ∈ V(G') such that d(u) < k then
5:             U ← original vertices contracted to u
6:             output G[U]
7:             Remove u from G'
8:         else
9:             Choose an edge (v, w) in G' at random
10:            Contract v and w in G'
```

Algorithm 4 Overall Algorithm

```
1: procedure K-EDGE-CONNECTED-COMPONENTS(G, k, t)
2:     φ_0 ← {G}
3:     for i = 1, 2, ..., t do
4:         φ_i ← {}
5:         for all G' ∈ φ_{i−1} do
6:             φ_i ← φ_i ∪ ContractAndCut(G', k)
7:     return φ_t
```

The reason behind such immediate contraction lies in the understanding that if the edge's weight is at least k, there's no way to separate the two vertices by a cut smaller than this weight. So contracting these vertices will not ruin any potential cut of size less than k and by contracting them, the chances of finding other cuts smaller than k increase.

The algorithm doesn't specify the number of iterations, which can lead to potential errors. However, it is suggested in [1] that by setting the number of iterations to $O(\log_2 n)$, the error probability can be reduced to as low as $\frac{1}{1000}$. In our tests, we carefully chose the number of iterations to accurately determine maximal-k-edge connected components.

Example 4. *Consider the graph depicted in Figure 1 with $k = 3$. Initially, we create G', a replica of G. During one iteration of contract and cut, suppose the randomly selected edge is (v_{10}, v_{11}), which is then contracted. This contraction entails removing the edge between v_{10} and v_{11}, and merging v_{10} and v_{11} into a supervertex. Since the sizes of the dictionaries of v_{10} and v_{11} are the same, we choose to add the neighbors of v_{10} to the dictionary of v_{11} and the remove v_{10} from the graph.*

Given that the set of edges in the graph is not empty, we proceed to select another edge. Suppose the chosen edge this time is (v_{11}, v_{12}). The graph after these contractions is given in Figure 3.3.

Yet again, since the set of edges remains non-empty, we pick a random edge. This

time, let us assume the chosen edge is (v_8, v_{12})*. After this contraction the degree of vertex* v_8 *is 2, which is less than* k*, so we extract the induced subgraph consisting of vertices contracted with* v_8*, namely* v_8, v_{10}, v_{11}*, and* v_{12} *from* G*. This subgraph, referred to as* $G1$*, emerges as one of the outputs from the first iteration. Subsequent to this,* v_8 *and its associated edges are removed from* G*. This process of picking and contracting random edges continues until there are no edges left in the graph.*

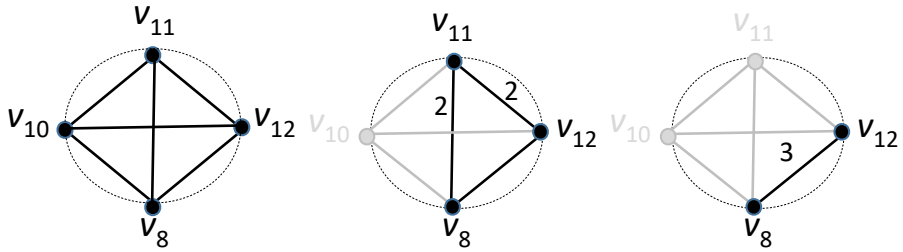


Figure 3.3: The figures illustrate the changes in the graph presented in Figure 1 following a series of contractions. We have visualized only the related vertices. Figure (a) illustrates the section of graph in 1 that we used for this example before contraction. Figure (b) shows the graph after contracting the edge between v_{10} and v_{11}, resulting in creating vertex v_{11}, which is formed by merging v_{10} and v_{11}. Figure (c) reveals the graph after a further contraction between v_{11} and v_{12}, leading to the creation of vertex v_{12}, which is obtained by merging v_{10}, v_{11}, and v_{12}.

Our contribution in this section is an optimization of the random contraction implementation, drawing inspiration from a graph representation technique presented in [6]. This array-based method provides an advantage during graph traversal: all the graph data resides in a contiguous block of memory. This arrangement accelerates the process compared to traditional adjacency lists.

In this structure, a graph is represented using four arrays. Central to this representation is the graph_head array. For every vertex v in V, graph_head(v) indicates the index of the starting neighbor in the value array. The value array keeps the actual neighbors of the vertices in the graph. Using the "next" array, we can identify the indices of subsequent neighbors.

When next$(i) = -1$, it indicates the end of that vertex's adjacency list. The "previous" array functions like next, but instead points to the preceding neighbor. The bidirectional aspect of the graph is addressed by the reverse array. For an edge (u, v) located at index i in value, "reverse" keeps its counterpart edge (v, u) in a manner that if $value(i) = v$, then $value(reverse(i))$ would represent u. So based on what we explained, to retrieve all neighbors of a vertex v, we start at graph_head(v) and keep iterating using the next array until reaching -1.

Example 5. *Refer to graph represented in Figure 3.1.a. The proposed data structure for this graph is illustrated in Figure 3.4. The graph-head* (v_0) *is 4, indicating that the first neighbor of* v_0 *is found at the 4th index of the value array, which is* v_3*. The index of the subsequent neighbor of* v_0 *can be determined using next*(4)*, which yields 2. This suggests that the next neighbor of* v_0 *is positioned at the 2nd index of the value array, corresponding to* v_2*. Following this pattern, next*(2) *is 0, pointing to the 8th index of the value array where* v_1 *is located. As next*(0) *equals -1, it signifies that* v_1 *is the last neighbor of* v_0*. Thus, the neighbors of* v_0 *are* v_1*,* v_2*, and* v_3*.*

When vertices u and v are contracted, we add v's neighbors to u's neighbor list. If a vertex x is a neighbor to both u and v, it will appear twice in u's list after the merge. If we were to store edge weights in a separate array, updating these weights during a merge would require checking if a neighbor of v is also a neighbor of u before adjusting the weight. This check takes $O(n)$ time for our structure.

Since keeping weights implies additional running time when using our proposed data structure, we do not maintain weights in our implementation. As such, we forego forced contractions, which depend on these weights and hence cannot be applied. However, empirical testing on diverse datasets reveals that our array-based implementation consistently outperforms the original algorithm based on dictionaries.

graph head

4	8	10	11

value

v1	v0	v2	v0	v3	v0	v2	v1	v3	v1	v3	v2

next

-1	-1	0	-1	2	-1	1	3	6	5	7	9

reverse

1	0	3	2	5	4	7	6	9	8	11	10

previous

2	6	4	7	-1	9	8	10	-1	11	-1	-1

Figure 3.4: Data Structure visualization For Graph In Figure 3.1.a

3.3 Early Merging And Splitting

The early merging and splitting algorithm, known as the MSK algorithm, was proposed by [37]. The algorithm determines the k-edge connected components within a graph by sequentially examining an ordered list of vertices. This order is established based on each vertex's connectivity within the graph. As the algorithm processes this list, it merges any two vertices exhibiting k-edge connectivity into a singular super-vertex. Conversely, if the vertex pair does not satisfy the condition of k-edge connectivity (there exist a cut of size less than k separating the two vertices), the edges of the cut are removed from the graph, and the graph is decomposed into two subgraphs. This procedure continues iteratively on the resultant subgraphs until every one of them qualifies as a k-edge connected component.

This method presents notable similarities to the approach of [6] described in section 3.1. Both strategies utilize the MAS procedure to find minimum s-t cuts. Additionally, in either approach, multiple vertex pairs can be merged in one MAS iteration using the early merging technique, as long as the connectivity between s and t remains at least k.

The key distinction lies in how the cuts are split. In `Decompose-LMS`, cuts are split after completing the list L for the MAS procedure. In contrast, in MSK, before inserting vertices into L during the MAS process, we check the weight between these vertices and those not in L. If this weight is below k, the graph splits into two subgraphs: one formed by the vertices in L and another from the original vertices that had merged with vertices in $V \setminus L$ in prior iterations. So, in MSK, as soon as the weight between vertices in L and vertices in $V \setminus L$ is less than k, we immediately split the cuts and decompose the graph into the two subgraphs.

The pseudocode for the MSK algorithm is presented in Algorithm 5, where we incorporate the linear data from Section3.1 for the MAS procedure into the original

algorithm.

The time complexity of the initial merge and split algorithm is denoted by $r \cdot t$, where t represents the time required to compute the order list L during the Mass process. In Algorithm 5, as we employ a linear data structure for the Mass procedure, the time complexity becomes $O(r \cdot m)$.

Example 6. *Consider a graph represented in Figure 2.1, with $K = 3$. Initially, L is an empty set. The traversal begins from vertex v_8, which is then added to L, making $L = [v_8]$. At this point, vertices v_7, v_9, v_{10}, v_{11}, and v_{12} all exhibit similar connectivity to L. From these, v_7 is chosen for inclusion in L. As we progress, vertices with the highest connectivity to L are consistently added. When L becomes $[v_8, v_7, v_9, v_{13}]$, v_4 becomes the next candidate, as its connectivity to the vertices in L, $w(v_4, L)$, is at least 3. This leads to v_4 merging with the last vertex in L, forming a super-vertex, u_1. Vertices originally linked to v_{13} or v_4 now connect with u_1. The process of adding to L continues until all V_0, which are the vertices in the graph, become empty. In the first iteration, since the weight between vertices in L and those outside L is always greater than K, the split is not applied. The outcome of this iteration is depicted in Figure 3.5.*

For the next iteration, traversal starts again from v_8, and L is reset to $[v_8]$. Here, the super-vertex u_4, comprising vertices v_{11} and v_{12} from Figure 2.1, emerges as the most connected to L. Upon its addition to L, v_{10} becomes the next candidate, leading to its merger with u_4, resulting in a new super-vertex, u_5. Given the existing edge connections and L, since the weight between vertices in L which are v_8, u_5 (u_5 is obtained by merging $\{v_{10}, v_{11}, v_{12}\}$) and the vertices that are not in L is less than 3, the graph undergoes decomposition, producing two subgraphs. One has vertices v_8 and u_5, while the other includes the rest. Due to the decomposition, there can be discrepancies in k-edge connectivity, especially for vertices outside L. These vertices are then reverted

Algorithm 5 MSK-Linear Data Structure

Require: A graph $G = \langle V, E \rangle$ and an integer k
Ensure: k-edge connected subgraphs (R) of G

Initialize a queue P_G consisting of a single graph G
while $P_G \neq$ NULL **do**
    $G_0 \leftarrow \text{pop}(P_G)$; $G_0 = (V_0, E_0)$
    Initialize the Linear Data Structure
    Initialize a vertex order $L = \emptyset$, add v_0 of V_0 into L
    for $s \in$ Neighbors(v_0) **do**
        update-key for s
    while $V_0 \neq$ NULL **do**
        if $w(V_L, V_0 \setminus V_L) \geq k$ **then**
            insert u into L orderly, $u =$ extract_max
            for $s \in$ Neighbors(u) **do**
                update-key for s
            if $w(V_L \setminus u, u) \geq k$ **then**
                G_1 is obtained by merging vertex u and vertex v
                before u in L
                $G_0 \leftarrow G_1$; $G_0 = (V_0, E_0)$
                if $|V_0| = 1$ **then**
                    the super vertex in V_0 is restored to normal
                    vertices
                    the subgraph constructed by vertices of
                    $\text{Org}(V_0)$ is inserted into R
                $V_0 \leftarrow V_0 \setminus V_L$
        else
            G_0 is decomposed into two separated subgraphs,
            one of which is constructed by vertices in V_L
            and the other one by restored supervertices in $V_0 \setminus V_L$.
            if $|V_L| = 1$ **then**
                The super vertex in V_L is restored to normal
                vertices
                The subgraph constructed by vertices of $\text{Org}(L)$
                is inserted into R
            else
                $G[V_L]$ is inserted in queue P_G
                $G[\text{Org}(V_0 \setminus V_L)]$ is inserted in queue P_G
            Initialize the Linear Data Structure
            set $L = \emptyset$, add v_0 of V_0 into L
            for $s \in$ Neighbors(v_0) **do**
                update-key for s
            $V_0 \leftarrow V_0 \setminus v_0$
    if graph G_0 has more than one vertex **then**
        insert G_0 back to P_G
return R

to their original state. This merge-and-split approach continues for every subgraph in the queue until it's exhausted. The end result is the trio of edge-connected subgraphs, $G1, G2$, *and* $G3$, *visible in Figure 2.1.*

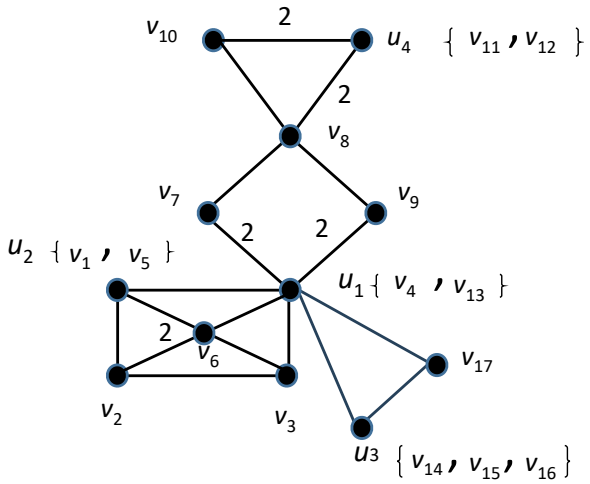


Figure 3.5: This figure illustrates the graph from Figure 2.1 at the end of the first iteration of the MSK algorithm. The weights of the edges are indicated next to the edges.

Incorrect claim. In the MSK paper, the authors view the algorithm of [6], as detailed in the graph decomposition section, as an approximation rather than an exact method for identifying k-edge-connected components. This interpretation stems from the fact that, during the `Decompose-LMS` algorithm, there exists a possibility for two vertices to merge provided their connectivity is at least k in the input graph. Yet, when certain cuts are removed in subsequent iterations, the connectivity between these vertices might drop below k.

Consider the example given in Section 3.1 (Example 3). Vertices v_4 and v_{13} are merged into a single super vertex. However, after eliminating the cut between v_8 and v_7 (specifically, the edges (v_8, v_7) and (v_8, v_9)), v_4 and v_{13} lose their k-connectivity. Since the vertices have already been merged, disentangling them becomes infeasible. As a result, post the initial `Decompose-LMS` execution, G_1 and $G_2 \cup G_3$ could be

deemed as connected components.

Authors of [37] incorrectly deduced that the results from a singular `Decompose-LMS` invocation amounted to the k-edge-connected component for the graph decomposition procedure as illustrated in Section 3.1. This misinterpretation underpins their labeling of the graph decomposition as an approximate algorithm. However, the true k-edge-connected components are extracted from Algorithm 1 in Section 3.1. Notably, `Decompose-LMS` is recursively executed until all components achieve k-connectivity.

Upon unifying the implementation environments of both algorithms and integrating the heap data structure from `Decompose-LMS` into MSK to optimize the MAS procedure, our comprehensive tests favored graph decomposition over MSK. We delve into the details of these findings in the experiments section.

3.4 Local Cut Detection

Chechik et al. in [7] present yet another algorithm that computes maximal k-edge-connected components in directed graphs [7]. A directed graph is k-edge connected if it is strongly connected whenever fewer than k edges are removed. Notably, while their algorithm targets directed graphs, it can also be used for undirected graphs. This is achieved by representing an undirected graph as a directed graph with the same vertex set, where each undirected edge is replaced by two bidirectional edges.

The main idea of this method is to identify a small subgraph with at most $\sqrt{m}$ edges that's distinct from the rest of the graph. This is done by performing DFS traversals that, when starting from some vertex, traverse mostly the edges within this small subgraph and a few edges outside it. This subgraph is considered "well-separated" because it isn't k-edge-connected to the other parts of the graph. After identifying it, the edges connecting this subgraph to the rest of the graph are removed.

This step is referred to as `local cut detection`. The local cut detection is repeated until no cut in the graph is smaller than k. More specifically, a `local cut detection` identifies a k-edge-out component of a vertex u, which is a subgraph that contains u and has no more than k edges extending from the subgraph to the remainder of the graph. The procedure to compute the k-edge-out components is as follows.

1. Start with a vertex u.
2. Initialize a depth-first search (DFS) from u.
3. Begin a loop that runs until $2k' + 1$, where $k' = k - 1$.
4. During each iteration, allow the DFS to extend by $\delta + 1$ edges, but only consider forward edges in the DFS tree, where δ starts initially as $\sqrt{m}$.
5. If the extension encompasses at most δ edges, the vertices visited during all extensions (denoted as F) are considered as part of the k-edge-out component.
6. If the recursion depth $\leq k'$:
    - Determine the nearest common ancestor (NCA) of the visited vertices from the extension.
    - Reverse the edges on the DFS tree starting from the NCA.
    - Apply the algorithm recursively on the modified graph, G'.
7. Continue the process until the recursive steps produce a non-empty result.

Similarly, a k-edge-in component is computed in the same manner but on the inverse graph.

The algorithm to compute k-edge-connected components begins with a given graph and a list L, which initially contains the vertices of the graph. After initializing the list L and setting the initial number of edges in the graph as m, the

algorithm checks if the graph has a cut of less than k edges. If not, the graph is returned as a k-edge-connected component. Otherwise, a loop starts and continues until L is not equal to the empty set and the graph contains more than $2k\sqrt{m}$ edges. In each iteration, the k-edge-out component and k-edge-in component are calculated for a vertex u extracted from list L. If either of these is not equal to the empty set, the algorithm removes the edges with endpoints in the result of the k-edge-out or k-edge-in and continues. After the loop, the strongly connected components (SCCs) of the resulting graph are calculated.

A set U is initialized as an empty set. For each SCC, the algorithm calculates a $k-1$-cut set if it exists, removes the $k-1$-cut set, and computes the SCCs of the resulting graph. Edges between the SCCs are removed and their endpoints are saved in a list. For each SCC, a list L' is created and populated with the vertices of the SCC that are in the saved list of endpoints. The algorithm then recursively processes the SCC and L', and unites the results with U.

The time complexity of the k-edge-connected components algorithm is $O((2k)^{k+1} \times \delta)$, where k is a constant representing the edge connectivity and $\delta = \sqrt{m}$. This complexity if broken down as follows.

- The DFS search takes time $(2k'+1) \times (\delta+1)$ and makes at most $2k'+1$ recursive calls. The recursion depth is explicitly bounded by $k = k' + 1$.

- The total number of recursive calls is $(2k' + 1)^{k'+1}$. Multiplying this with the time taken for each call (DFS search time), we get the overall time complexity of $O((2k)^{k+1} \times \delta)$.

In the paper, k is treated as a constant; hence, the factor $(2k)^{k+1}$ is omitted from the analysis. The overall complexity stands at $O(m\sqrt{m}\log n)$ when this factor is disregarded. Nonetheless, in real-world applications, k can be quite large, ranging

between 30 to 60 for actual graphs. For such magnitudes, the algorithm presented in [7] becomes impractical.

For instance, in the soc-epinions graph used in our experiments, the maximum k value is 67. This translates to $(2k)^{2k+1}$ being roughly 290,000,000. As a result, computing the 67-connected components for this graph using the current algorithm is impractical for real-world scenarios.

Chapter 4

Experiments

Our study conducts a comparative assessment of selected algorithms, emphasizing their running times as the primary metric. We analyzed the efficiency of each algorithm under standardized conditions. All algorithms were implemented in Java to ensure consistency. The source code can be accessed in our GitHub repository[1]. We performed our experiments on Compute Canada's Cedar5, a high-performance computing cluster equipped with dual 6-core 2.10 GHz Intel Xeon CPUs. Each test was allocated 32GB RAM.

We evaluated our algorithms on eight real graphs. Real graphs, derived from actual data sources, contain inherent structures and patterns that represent real-world scenarios. All the graphs are obtained from the Stanford SNAP library2, and detailed descriptions of these graphs can also be found there. Sizes of these graphs are shown in Table 4.1. By varying the sizes of these graphs, we can evaluate our algorithms across different graph structures, from small to large. For clarity, we've also summarized the main details of the algorithms we studied in Table 4.2.

[1] `https://github.com/Haniehsadri/KECC`

Data Set	# Vertices	# Edges
bird	129	954
feather-lastfm-social	7,624	27,806
ego-Facebook	4,039	88,234
feather-deezer-social	28,281	92,752
musae-git	37,700	289,003
soc-epinions	75,879	508,837
com-DBLP	317,080	1,049,866
amazon	262,111	1,234,877

Table 4.1: Datasets

Algorithm	Abbr	Section	Complexity
Graph Decomposition	GD	3.1	$O(h \cdot l \cdot m)$
Random Contraction	RC	3.2	$O(m \cdot \log n)$
Random Contraction Flat	RCF	3.2	$O(m \cdot \log n)$
Early Merge and Split	MSK	3.3	$O(r \cdot m)$
Local Cut Detection	LCD	3.4	$O(m \cdot \sqrt{m} \cdot \log n)$

Table 4.2: The table lists abbreviations, associated sections, and time complexities of algorithms. For time complexities: m is the edge count, n is the vertex count, h represents the repetition count for Decompose_LMS in Algorithm 1 (typically a small integer for real-world graphs), l is the iteration count in Decompose_LMS (usually less than the number of vertices), and r indicates iteration counts in the MSK algorithm.

4.1 Small Graphs

The small datasets that we considered are bird with 958 edges, feather-lastfm-social with 27,806 edges, ego-Facebook with 88,234, and feather-deezer-social with 92,752. We plot the run times achieved by RC, RCF, GD, and MSK on all the small datasets. However, for LCD, we only recorded its runtime for the bird dataset. This algorithm was not scalable for the other datasets; it failed to produce results even after an extended period of 48 hours.

From Figure 4.1, it is evident that the LCD is significantly slower than the other algorithms. While most algorithms process the bird dataset in a negligible time (close to 0 seconds), on average, the LCD algorithm takes 1500 seconds to process this dataset. From Figures 4.1.b, 4.1.c, and 4.1.d, it is clear that GD requires less time to complete and outperforms the other algorithms in all instances. RCF is faster than both the RC and MSK. When comparing RC to MSK for smaller data sets, there are situations where RC performs better, and in other instances, the MSK demonstrates a faster runtime. From all the plots in 4.1 it can be seen that the GD outperforms

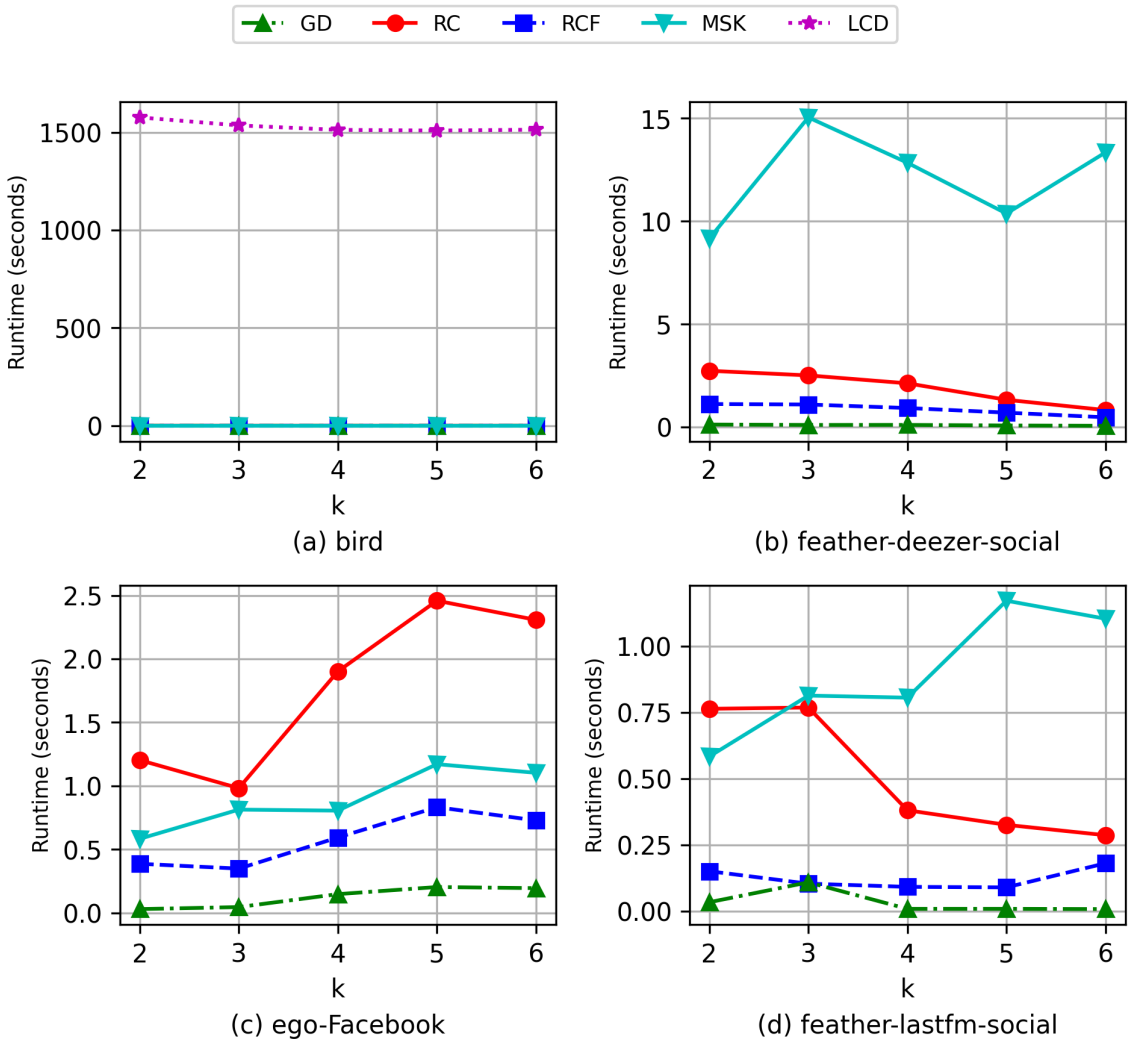


Figure 4.1: Performance Analysis on Small Datasets: The figure compares the runtime of various algorithms on small datasets. It underscores the notable efficiency of GD and RCF while illustrating the pronounced slower runtime of LCD.

other algorithms for small datasets.

4.1.1 Medium and Large Graphs

For medium and large graphs, we considered several datasets: the musae-github graph with 289,003 edges, soc-epinions with 508,837 edges, com-DBLP which contains 1,049,866 edges, and the sizable amazon0302 graph which comprises 1,234,877 edges. We plotted the run times achieved by the four algorithms and the results can be observed in Figure 4.2.

In evaluations involving medium and large datasets, the GD algorithm consistently outperforms the other algorithms, much like its performance with smaller datasets. The RCF algorithm outperforms both the MSK and the RC algorithms in terms of efficiency. For medium-sized graphs, MSK yields better results when compared to RC. However, when we transition to larger graphs, MSK starts to falter in terms of scalability, a trend that's evident in Figures 4.2.c and 4.2.d. We mention here that we are the first to be able to run our MSK implementation to large datasets. The original authors were only able to run up to the Soc-Epinions dataset.

A similar trend observed across plots in Figure 4.2 is that, for most cases, the processing times of all four algorithms decrease as k increases. As k becomes larger, the graph, after removing all vertices with degrees less than k, becomes smaller, leading to faster execution of all algorithms. However, this trend might vary for certain graphs. For instance, with the Amazon dataset (shown in Table 4.5), as k increases, the number of components also increases significantly, necessitating a considerable increase in the number of iterations for RC. It is only when k goes from 5 to 6 that the number of components drops drastically from 2653 to 32. At this point the running time decreases significantly.

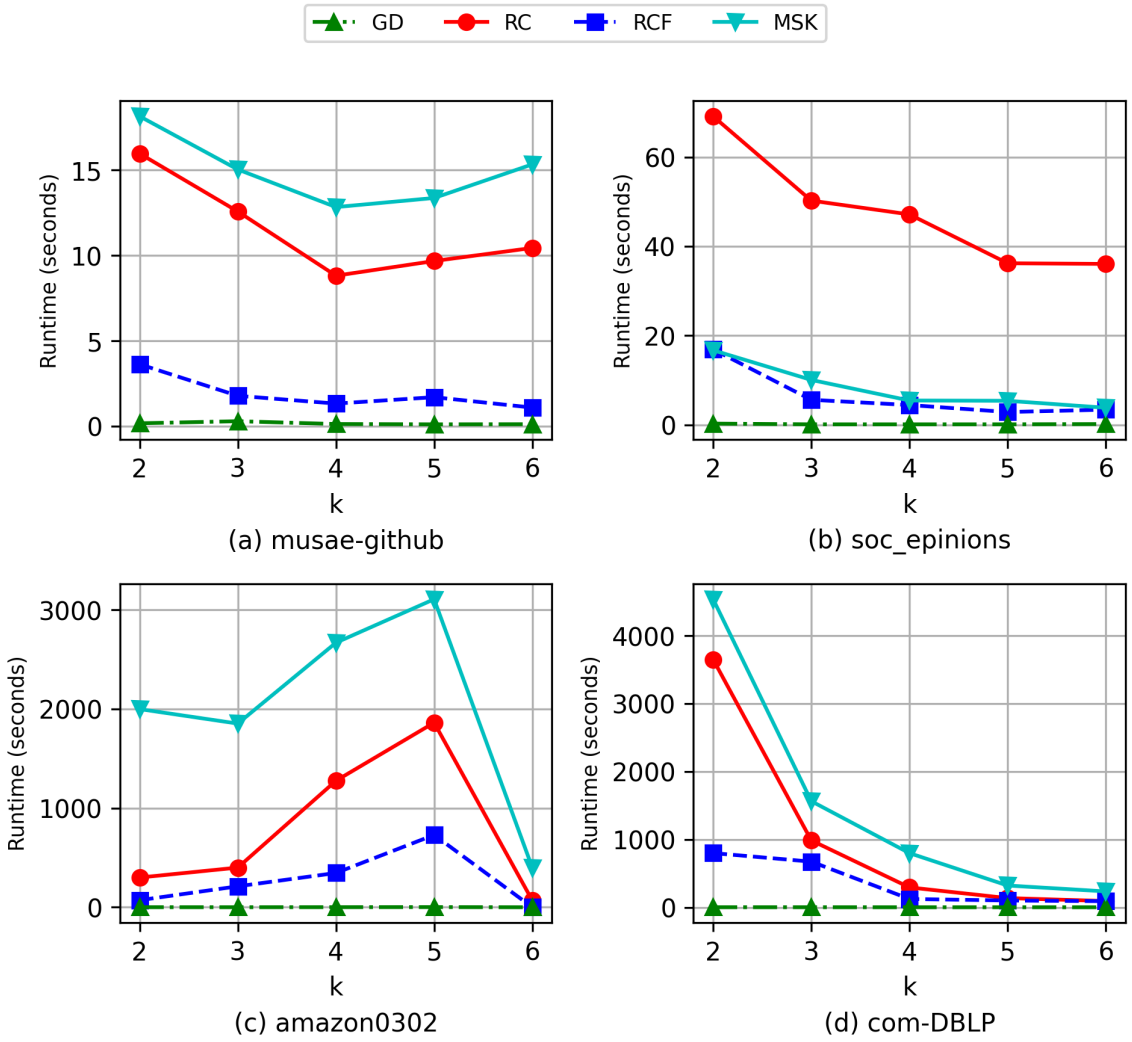


Figure 4.2: Performance Analysis on Medium to Large Datasets: The figure presents the runtime performance of four algorithms across varying graph sizes, emphasizing the consistent efficiency of GD and the relative advantages of RCF over RC. Observe that for amazon, the biggest dataset, as k increases, the runtime increases as well up until $k = 5$, then it decreases drastically for $k = 6$. This is because the number of k-connected components increases with k, up to $k = 5$, then it becomes very small for $k = 6$ (see Table 4.5).

Another analysis we perform is the evaluation of the performance of each algorithm on datasets of varying sizes, assessing how their efficiency changes as the data size transitions from small to medium, and from medium to large (see Figures 4.3 to 4.7). We demonstrated the experiment with k ranging from 2 to 6.

Starting with the GD algorithm, our experiments show that it performs efficiently and consistently across graphs of different sizes. Even when the graph size increases, the increase in time is not significant. As can be seen in Figures 4.3 to 4.7, the running time of GD remains consistently low whether the data size changes from small to medium or from medium to large.

Moving on to the RC algorithm, it performs well for small to medium-sized graphs. However, its efficiency decreases for larger graphs, such as the Amazon dataset. Figures 4.3 to 4.7 show a noticeable increase in the running time when transitioning from medium to large-sized graphs. The RCF algorithm performs well for both small and medium datasets. The efficiency of this algorithm slightly changes when the data size increases from medium to large.

Examining the MSK algorithm, it operates efficiently for small to medium datasets. Yet, for larger datasets, such as amazon, its performance significantly drops. There's a distinct increase in its running time when the dataset size shifts from medium to large.

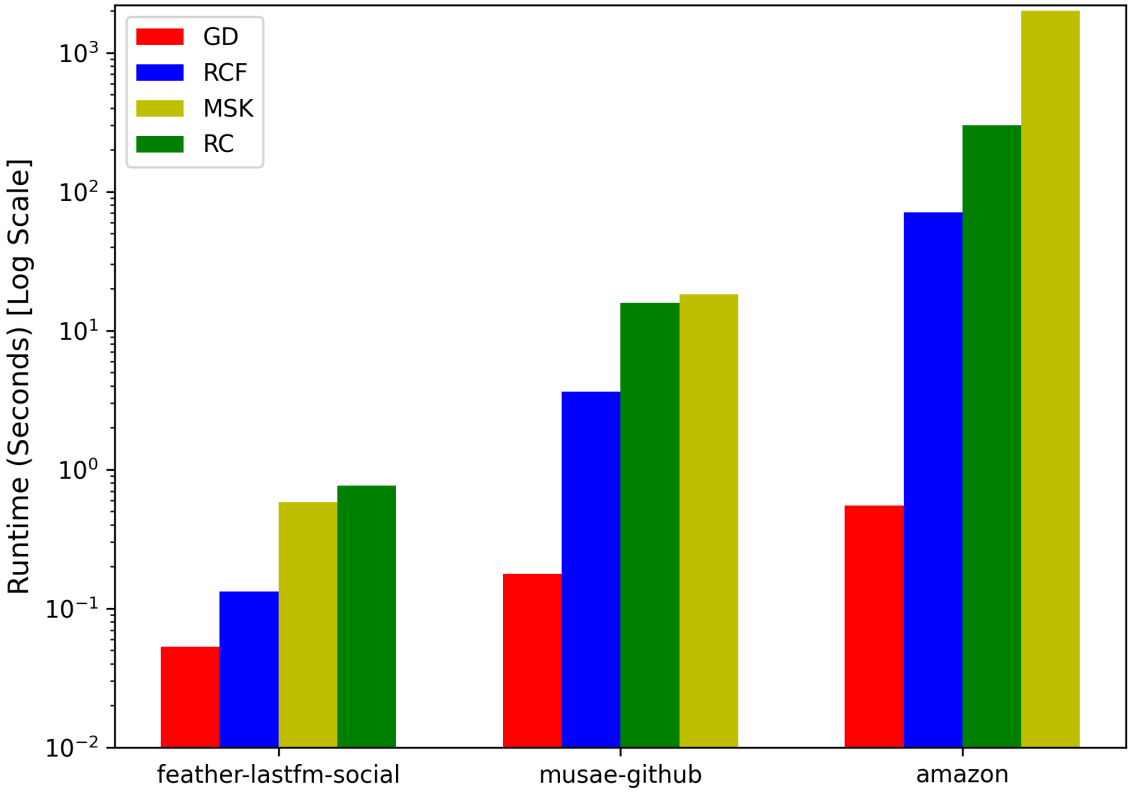


Figure 4.3: Comparing running times for GD, RC, RCF, and MSK on three largest datasets for $k = 2$: The figure underscores the consistent efficiency of GD, the waning performance of the RC and MSK on larger datasets, and the subtle transition in efficiency of RCF from medium to large datasets.Upon conducting experiments with various values of k, we found that the results conformed to the same trend exhibited when $k = 2$.

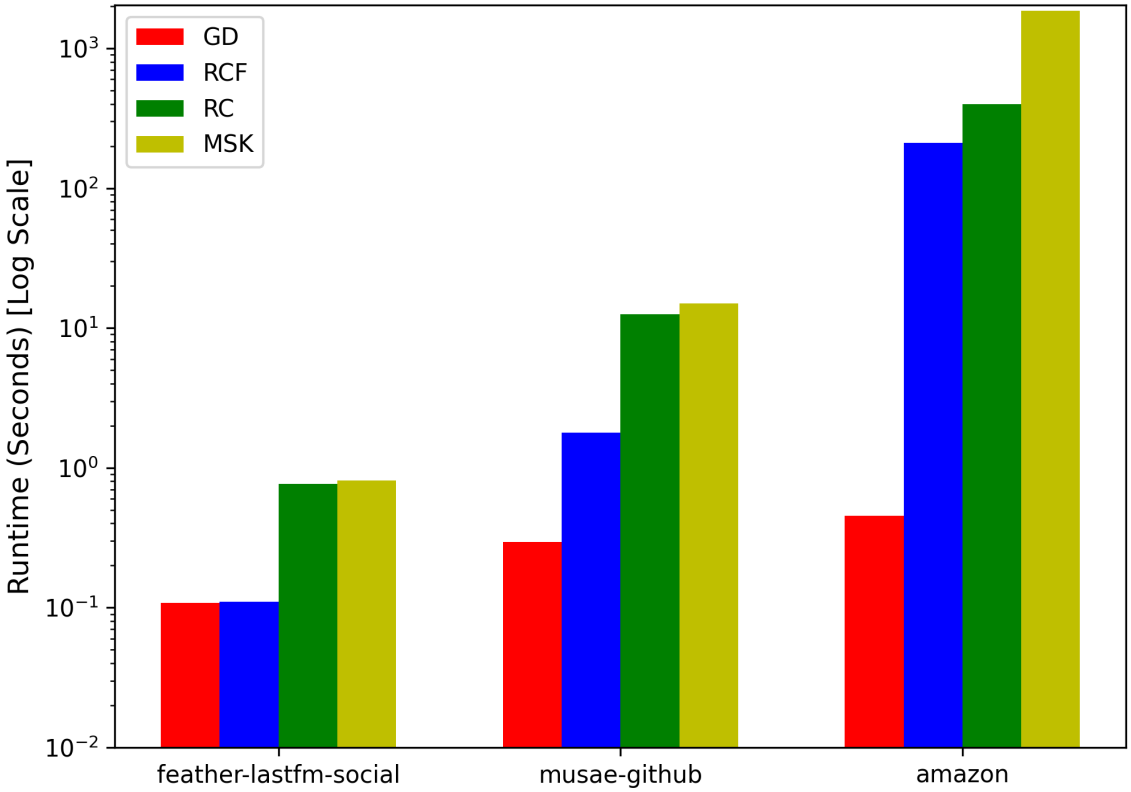


Figure 4.4: Comparing running times for GD, RC, RCF, and MSK on three largest datasets for $k = 3$: The figure underscores the consistent efficiency of GD, the waning performance of the RC and MSK on larger datasets, and the subtle transition in efficiency of RCF from medium to large datasets.Upon conducting experiments with various values of k, we found that the results conformed to the same trend exhibited when $k = 3$.

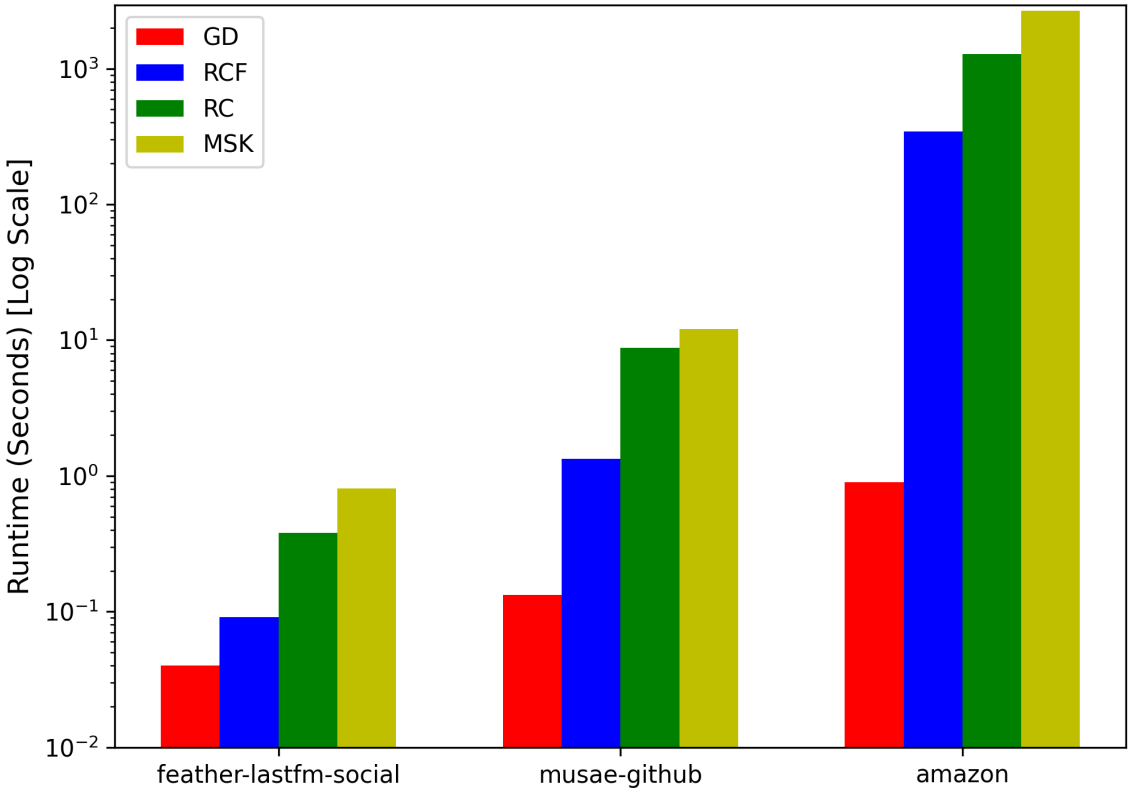


Figure 4.5: Comparing running times for GD, RC, RCF, and MSK on three largest datasets for $k = 4$: The figure underscores the consistent efficiency of GD, the waning performance of the RC and MSK on larger datasets, and the subtle transition in efficiency of RCF from medium to large datasets.Upon conducting experiments with various values of k, we found that the results conformed to the same trend exhibited when $k = 4$.

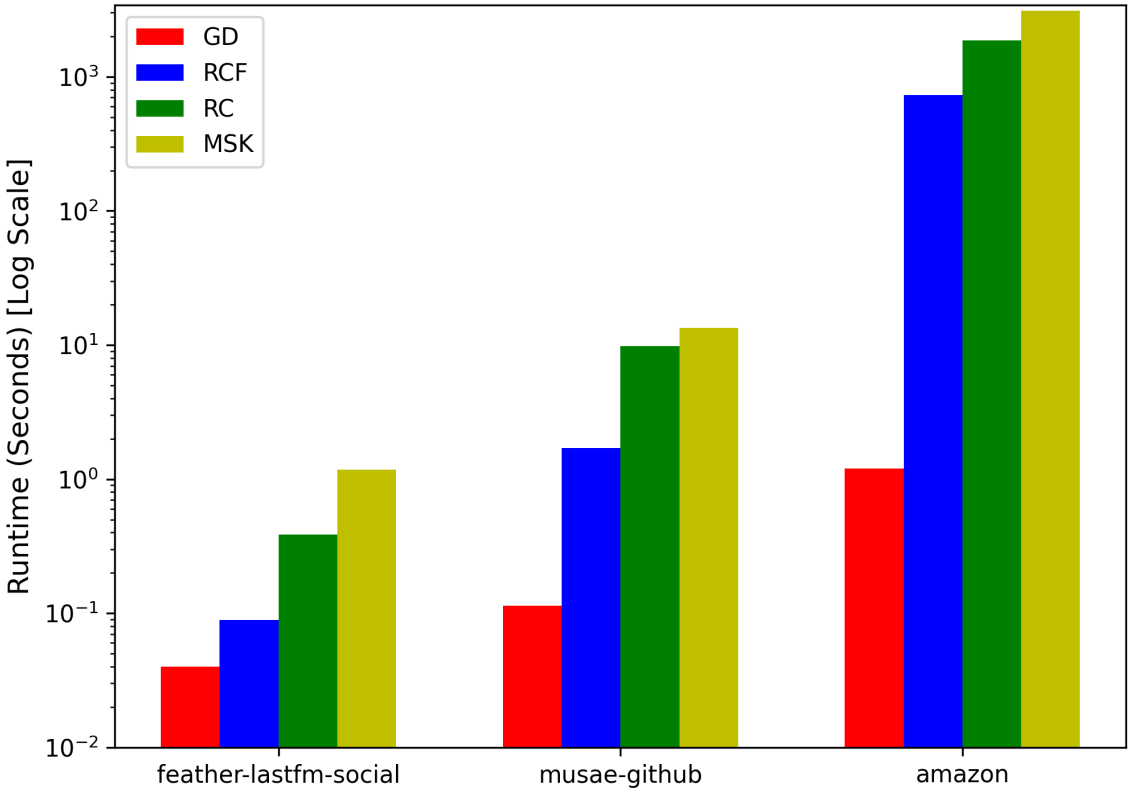


Figure 4.6: Comparing running times for GD, RC, RCF, and MSK on three largest datasets for $k = 5$: The figure underscores the consistent efficiency of GD, the waning performance of the RC and MSK on larger datasets, and the subtle transition in efficiency of RCF from medium to large datasets.Upon conducting experiments with various values of k, we found that the results conformed to the same trend exhibited when $k = 5$.

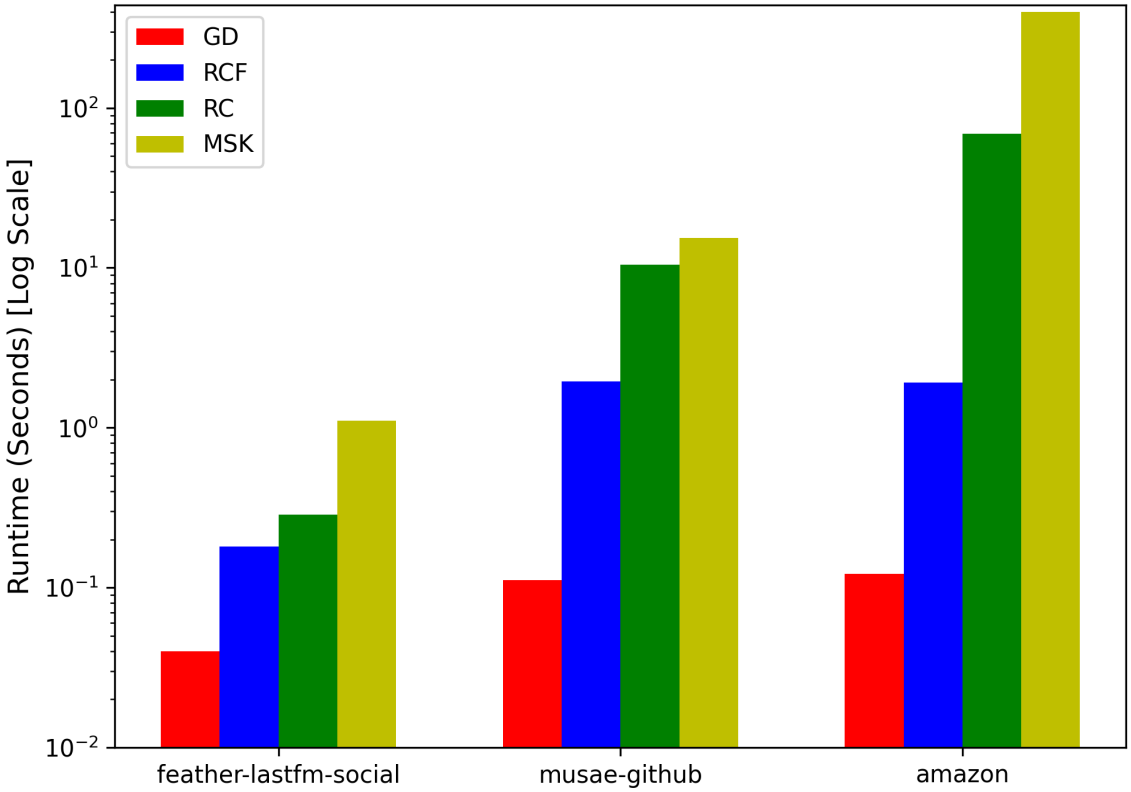


Figure 4.7: Comparing running times for GD, RC, RCF, and MSK on three largest datasets for $k = 6$: The figure underscores the consistent efficiency of GD, the waning performance of the RC and MSK on larger datasets, and the subtle transition in efficiency of RCF from medium to large datasets.Upon conducting experiments with various values of k, we found that the results conformed to the same trend exhibited when $k = 6$.

4.2 Evaluation of Optimization Techniques for RC

In this subsection, we evaluate the effectiveness of the optimization techniques introduced in Section 3.2 for the RC algorithm. The experiments conducted on various datasets underscore that the RCF method consistently outperforms the original RC approach in empirical tests–even without implementing forced contraction. To elucidate this distinction further, we examine both algorithms on the `amazon` dataset.

We emphasize that the components produced in each iteration of `RC` and `RCF` are connected, but they may not always be k-edge-connected. Therefore, multiple iterations with each algorithm are required to achieve the desired number of exact k-edge connected components. We recall that both RC and RCF are randomized algorithms, which carry a small (but non-zero) probability of failing to identify all the k-ecc components. We used the results from GD as a benchmark to determine the number of iterations needed for RC and RCF to correctly identify these components.

K	**Iter.**	**RC**	**RCF**	**GD**
2	1	5	6	6
	2	6	6	
3	1	2	2	2
	2	2	2	
4	1	1	1	1

Table 4.3: Number of extracted k-ecc's after each iteration for RC and RCF on musae-git.

In the plots illustrated in 4.1 and 4.2, we executed both the RC and RCF algorithms using the appropriate number of iterations. From the plots, it is evident that even though the RC needs fewer iterations to reach the desired number of components, RCF consistently outperforms it time-wise. As an example, let us consider the Amazon dataset from table 4.5, with $k = 4$. The RC algorithm identifies the

K	Iter.	RC	RCF	GD
2	1	73	58	87
	2	85	83	
	3	87	85	
	4	87	87	
3	1	14	12	21
	2	20	14	
	3	21	21	
4	1	8	8	10
	2	10	8	
	3	10	10	
5	1	5	5	8
	2	8	8	
6	1	6	3	7
	2	7	5	
	3	7	7	
7	1	2	2	3
	2	3	3	
	3	3	3	

Table 4.4: Number of extracted k-ecc's after each iteration for RC and RCF on soc-epinions.

k	**Iter.**	**RC**	**RCF**	**GD**
2	1	355	350	367
	3	367	363	
	6	367	367	
3	1	681	402	777
	5	756	741	
	7	777	763	
	11	777	777	
4	1	876	678	1287
	7	1287	1202	
	14	1287	1287	
5	1	5	5	2653
	5	2437	2326	
	14	2653	2337	
	16	2653	2653	
6	1	32	32	32

Table 4.5: Number of extracted k-ecc's after each iteration for RC and RCF on amazon. Also shown are the numbers of k-ecc's obtained by GD, which serve as ground truth numbers. Recall that RC and RCF are randomized algorithms with some small (but non-zero) probability of not being able to discover all the k-ecc's. Also recall that RCF forgoes the forced random contractions that RC does.

1,287 4-edge-connected components in just 7 iterations. In contrast, the RCF algorithm needs 14 iterations to compute them. Yet, the runtime of RCF, even with 14 iterations, stands at 346 seconds, whereas the RC, with its 7 iterations, takes a significantly longer 1,277 seconds.

4.3 Discussion

In theory, all the algorithms presented in Section 3 possess linear time, or near linear, in m, complexity. However our experiments show variations in their performance across different datasets. From our results, the GD algorithm is the most efficient in determining k-edge-connected components. On the other hand, the RC algorithm struggles, especially with larger graph sizes. Notably, after improving its data structures and obtaining RCF, its speed improved considerably, making it more suitable for larger networks. However, even after these improvements, the optimized version, while better than its predecessor, still lags behind the GD algorithm.

The MSK algorithm was further refined by incorporating the heap data structure from *GD*, reducing its time complexity from $O(m + n \cdot \log n)$ to $O(m \cdot h)$. However, even after these changes, it is outperformed by the GD and RCF algorithms for larger datasets. For smaller and medium-sized datasets, it performs better than the RC algorithm. But for larger networks, its efficiency decreases, and RC becomes more efficient.

Regarding the LCD algorithm its practical scalability is limited, deeming it suitable only for small datasets. The original researchers only assessed its time complexity without any empirical tests. Its practical application appears limited, demonstrating feasibility only for very small networks.

Chapter 5

Conclusions and Future Work

In this research, we conducted an in-depth exploration of four specialized algorithms designed for the detection of k-edge connected components. Each algorithm was clearly explained, with a detailed discussion of potential optimizations for some of them, accompanied by an analysis of their computational time complexities.

To ensure a fair comparison, we implemented all algorithms based on the same underlying principles. We subsequently conducted a series of empirical studies using a variety of datasets to evaluate the scalability and practical effectiveness of the algorithms. These experiments were crucial for benchmarking the real-world performance of each algorithm in relation to one another, with a special emphasis on their execution times when applied to actual graph structures. The results of our experiments indicate that although all algorithms possess a broadly similar theoretical time complexity, their practical performance varied significantly across different datasets.

The findings of this study open several possibilities for future research, which could further advance our understanding and capabilities in detecting and analyzing k-edge-connected components in graphs. Prospective future work could revolve around the following questions:

1. **Parallel and Distributed Computing:** In the context of parallel and distributed computing, considerable progress has been made in developing parallel and distributed versions of algorithms for the related problems of k-core and k-truss decompositions (cf. [19, 20, 25, 26, 33]). Nevertheless, the parallelization of k-edge connected components computation remains elusive. This identifies a fertile area for future research, particularly in investigating how these algorithms could be adapted to effectively leverage distributed computing resources. A pertinent research question that emerges is: How can algorithms for computing k-edge connected components be parallelized to harness the power of parallel and distributed systems, and what scalability considerations must be addressed for very large graphs?

2. **Probabilistic Graphs:** In the realm of probabilistic graphs (cf. [35, 12]), critical questions arise: How might we effectively extend the k-edge-connected components concept to accommodate the complexity of probabilistic graphs in social networks? Given the attention that probabilistic graphs have attracted–owing to their capacity to express the strength of connections in social interactions–what are the prospects and challenges in adapting techniques that have been successful in k-core, k-truss, and graph clustering (cf. [11, 10, 16]) to k-edge connected components? Could leveraging statistical methods, particularly those based on the Central Limit Theorem ([11, 16]), provide a key to unlocking efficient computation of k-edge connected components in probabilistic graphs? Exploring these questions could pave the way for breakthroughs in processing the intricate webs of connections that probabilistic graphs represent.

3. **Dynamic Graphs:** In the context of social networks, the scalability of algorithms to handle dynamic graphs–where connections appear and disappear over time–is of paramount importance [9, 24]. Dynamic graphs offer a truer repre-

sentation of the fluid social interactions that characterize these networks. While there have been advances in k-core and k-truss decompositions as well as graph summarization for these dynamic settings (cf. [24, 17, 15]), the adaptation of k-edge-connected components to such environments is still an open question. It prompts us to consider: What strategies might we employ to bridge this gap? How can we refine the k-edge connected components algorithms to cope with the continuous evolution of graph data? Investigating these issues is not only interesting from a theoretical point of view but also vital for crafting analytical tools that can keep pace with the dynamic nature of social networks.

Bibliography

[1] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. Linear-time enumeration of maximal k-edge-connected subgraphs in large networks by random contraction. In *CIKM*, pages 909–918, 2013.

[2] Balabhaskar Balasundaram, Sergiy Butenko, and Illya V Hicks. Clique relaxations in social network analysis: The maximum k-plex problem. *Operations Research*, 59(1):133–142, 2011.

[3] Albert-László Barabási and Zoltán N Oltvai. Network biology: understanding the cell's functional organization. *Nature Reviews Genetics*, 5(2):101–113, 2004.

[4] Sergey Brin and Lawrence Page. Anatomy of a large-scale hypertextual web search engine. *Computer networks and ISDN systems*, 30(1-7):107–117, 1998.

[5] Lijun Chang and Zhiyi Wang. A near-optimal approach to edge connectivity-based hierarchical graph decomposition. *The VLDB Journal*, pages 1–23, 2023.

[6] Lijun Chang, Jeffrey Xu Yu, Lu Qin, Xuemin Lin, Chengfei Liu, and Weifa Liang. Efficiently computing k-edge connected components via graph decomposition. In *SIGMOD*, pages 205–216, 2013.

[7] Shiri Chechik, Thomas Dueholm Hansen, Giuseppe F. Italiano, Veronika Loitzenbauer, and Nikos Parotsidis. Faster algorithms for computing maximal 2-

connected subgraphs in sparse directed graphs. In *SODA*, pages 1900–1918, 2017.

[8] Daniel Delling, Peter Sanders, Dominik Schultes, and Dorothea Wagner. Engineering route planning algorithms. In *Algorithmics of large and complex networks: Design, analysis, and simulation*, pages 117–139. Springer, 2009.

[9] David Eppstein, Zvi Galil, and Giuseppe F Italiano. Dynamic graph algorithms. *Algorithms and theory of computation handbook*, 1:9–1, 1999.

[10] Fatemeh Esfahani, Mahsa Daneshmand, Venkatesh Srinivasan, Alex Thomo, and Kui Wu. Scalable probabilistic truss decomposition using central limit theorem and h-index. *Distributed Parallel Databases*, 40(2-3):299–333, 2022.

[11] Fatemeh Esfahani, Venkatesh Srinivasan, Alex Thomo, and Kui Wu. Efficient computation of probabilistic core decomposition at web-scale. In *EDBT*, pages 325–336, 2019.

[12] Fatemeh Esfahani, Venkatesh Srinivasan, Alex Thomo, and Kui Wu. Nucleus decomposition in probabilistic graphs: Hardness and algorithms. In *ICDE*, pages 218–231, 2022.

[13] Santo Fortunato. Community detection in graphs. *Physics reports*, 486(3-5):75–174, 2010.

[14] Zvi Galil and Giuseppe F Italiano. Reducing edge connectivity to vertex connectivity. *ACM Sigact News*, 22(1):57–61, 1991.

[15] Mahdi Hajiabadi, Venkatesh Srinivasan, and Alex Thomo. Dynamic graph summarization: Optimal and scalable. In *IEEE International Conference on Big Data (Big Data)*, pages 545–554. IEEE, 2022.

[16] Joseph Howie, Venkatesh Srinivasan, and Alex Thomo. Scaling up structural clustering to large probabilistic graphs using lyapunov central limit theorem. *Proceedings of the VLDB Endowment*, 16(11):3165–3177, 2023.

[17] Xin Huang, Hong Cheng, Lu Qin, Wentao Tian, and Jeffrey Xu Yu. Querying k-truss community in large and dynamic graphs. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 1311–1322, 2014.

[18] Hawoong Jeong, Sean P Mason, Albert-László Barabási, and Zoltán N Oltvai. Lethality and centrality in protein networks. *Nature*, 411(6833):41–42, 2001.

[19] Humayun Kabir and Kamesh Madduri. Parallel k-core decomposition on multicore platforms. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1482–1491. IEEE, 2017.

[20] Humayun Kabir and Kamesh Madduri. Parallel k-truss decomposition on multicore systems. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2017.

[21] David R Karger. Global min-cuts in rnc, and other ramifications of a simple min-cut algorithm. In *SODA*, pages 21–30, 1993.

[22] Wissam Khaouid, Marina Barsky, S. Venkatesh, and Alex Thomo. K-core decomposition of large networks on a single pc. *PVLDB*, 9(1):13–23, 2015.

[23] Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. Kronecker graphs: An approach to modeling networks. *The Journal of Machine Learning Research*, 11:985–1042, 2010.

[24] Rong-Hua Li, Jeffrey Xu Yu, and Rui Mao. Efficient core maintenance in large dynamic graphs. *IEEE Transactions on Knowledge and Data Engineering*, 26(10):2453–2465, 2013.

[25] Amir Mehrafsa, Sean Chester, and Alex Thomo. Vectorising k-core decomposition for gpu acceleration. In *32nd International Conference on Scientific and Statistical Database Management*, pages 1–4, 2020.

[26] Amir Mehrafsa, Sean Chester, and Alex Thomo. Vectorising k-truss decomposition for simple multi-core and simd acceleration. In *13th International Conference on Information, Intelligence, Systems & Applications (IISA)*, pages 1–6. IEEE, 2022.

[27] Alan Mislove, Massimiliano Marcon, Krishna P Gummadi, Peter Druschel, and Bobby Bhattacharjee. Measurement and analysis of online social networks. In *IMC'07*, pages 29–42, 2007.

[28] Hiroshi Nagamochi and Toshihide Ibaraki. A linear time algorithm for computing 3-edge-connected components in a multigraph. *Japan journal of industrial and applied mathematics*, 9:163–180, 1992.

[29] Hiroshi Nagamochi and Toshimasa Watanabe. Computing k-edge-connected components of a multigraph. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 76(4):513–517, 1993.

[30] MEJ Newman. The structure and function of complex networks. *SIAM review*, 45(2):167–256, 2003.

[31] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. In *Stanford InfoLab*, 1999.

[32] Norman J Pullman. Clique coverings of graphs—a survey. In *Combinatorial Mathematics X: Proceedings of the Conference held in Adelaide, Australia, August 23–27, 1982*, pages 72–85. Springer, 2006.

[33] Yudi Santoso, Xiaozhou Liu, Venkatesh Srinivasan, and Alex Thomo. Four node graphlet and triad enumeration on distributed platforms. *Distributed and Parallel Databases*, 40(2-3):335–372, 2022.

[34] S. Seidman. Network structure and minimum degree. *Social networks*, 5(3):269–287, 1983.

[35] M. Simpson, V. Srinivasan, and A. Thomo. Reverse prevention sampling for misinformation mitigation in social networks. In *Proc. of the 23rd International Conference on Database Theory (ICDT'20)*, pages 24:1–24:18. Schloss Dagstuhl-Leibniz-Zentrum fur Informatik, 2020.

[36] Victor Spirin and Leonid A Mirny. Protein complexes and functional modules in molecular networks. *Proceedings of the National Academy of Sciences*, 100(21):12123–12128, 2003.

[37] Heli Sun, Jianbin Huang, Yang Bai, Zhongmeng Zhao, Xiaolin Jia, Fang He, and Yang Li. Efficient k-edge connected component detection through an early merging and splitting strategy. *Knowledge-Based Systems*, 111:63–72, 2016.

[38] Yung H Tsin. Yet another optimal algorithm for 3-edge-connectivity. *Journal of Discrete Algorithms*, 7(1):130–146, 2009.

[39] J. Wang and J. Cheng. Truss decomposition in massive networks. *PVLDB*, 5(9):812–823, 2012.

[40] Tianhao Wang, Yong Zhang, Francis YL Chin, Hing-Fung Ting, Yung H Tsin, and Sheung-Hung Poon. A simple algorithm for finding all k-edge-connected components. *Plos one*, 10(9):e0136264, 2015.

[41] Jian Wu, Alison Goshulak, Venkatesh Srinivasan, and Alex Thomo. K-truss decomposition of large networks on a single consumer-grade machine. In *ASONAM*, pages 873–880, 2018.

[42] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems*, 42(1):181–213, 2015.

[43] R. Zhou, C. Liu, J. X. Yu, W. Liang, B. Chen, and J. Li. Finding maximal k-edge-connected subgraphs from a large graph. In *EDBT*, pages 480–491, 2012.