Efficient Computation of k-Edge Connected Components: An Empirical Analysis

Hanieh Sadri[1], Venkatesh Srinivasan[2], and Alex Thomo[1]

[1] University of Victoria, British Columbia, Canada,
haniehsadri77@gmail.com
thomo@uvic.ca
[2] Santa Clara University, California, USA,
vsrinivasan4@scu.edu

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 as 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: social networks · community detection · graph algorithms · k-edge connected components · empirical analysis.

1 Introduction

Graphs have become increasingly important in today's world due to their ability to capture complex relationships and provide valuable insights [19]. 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 [16], web searches [20], biochemistry [12], biology [2], and road network mapping [7]. Given their widespread use and significance, a lot of research is being conducted to analyze graph data [15].

Identifying communities within graphs, which are essentially clusters of densely connected vertices, is a vital concept due to its wide-ranging applications [10]. In social networking platforms, community detection can be leveraged for friend recommendations, targeted social campaigns, and advertising [28]. Within protein-protein interaction networks, community detection can be applied to recognizing

proteins with similar functionality [22]. 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 [3].

One prevalent method for identifying such clusters and communities is by computing k-edge connected components (cf. [29,1,5,23,6]). 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. [21,14,9]) or k-truss (cf. [25,27,8]) components, k-edge connected components often give well-connected communities that k-core and k-truss fail to discover (cf. [1] for more details).

Our work [3]provides a detailed exploration of four main algorithms for computing k-edge connected components. We implement each of these algorithms, compare their performance, and suggest optimization strategies when applicable.

The first algorithm we consider, presented in [5], 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 [23], which proposes an improvement to the decomposition algorithm of [5]. Its core idea is to combine vertices that meet the k-connectivity requirement. Lastly, we study an algorithm, given in [6], 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.

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 [5] 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.

[3] The full version of this paper is available here.

4. We highlight and correct inaccurate claims made by [23]. Contrary to their claims, we demonstrate that their proposed improvements do not perform as effectively in practice.
5. We provide the first implementation of an algorithm presented in [6] 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.

2 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. *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 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 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.*

3 Related Work

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 [5,23,1]. 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 [6,29].

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 [11,17,18,24].

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 [4] 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 [26] 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 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.

4 Algorithms

In this section, we present a thorough analysis of algorithms used to identify k-edge connected components in graphs. Moreover, we shed light on specific enhancements we have implemented to boost the performance of some of these algorithms. Among the studied methods, the graph decomposition algorithm emerged as the most effective for determining k-edge connected components.

4.1 Graph Decomposition Algorithm

An important algorithm, which we refer to as Graph Decomposition (GD), for identifying k-edge connected components was introduced by Chang et al. in [5]. 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

Input: A graph $G = (V, E)$ and an integer k.
Output: k-edge connected components of G.
Initialize a queue Q_g with the graph G as its sole member.
for each subgraph g in Q_g **do**
    $\phi_k(g) \leftarrow \text{Decompose}(g, k)$;
    if $\phi_k(g)$ consists solely of one subgraph **then**
        Output $\phi_k(g)$ as a k-edge connected component;
    else
        Enqueue all subgraphs of $\phi_k(g)$ 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 each 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.

An important part of the GD algorithm is creating several connected subgraphs by removing the edges in a cut of G with a 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 of 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, $\text{Head}[x] = v$, 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 1. Refer to Figure 1 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 $y > p_0$.

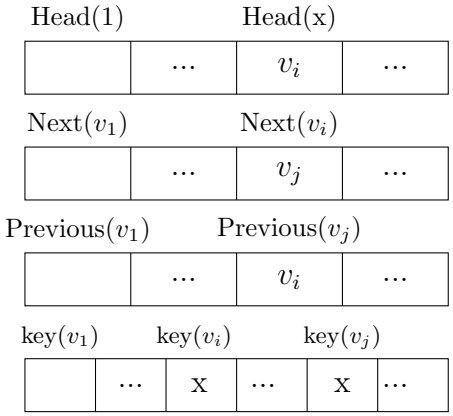

Fig. 1: Data structure

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.

An algorithm called `Decompose-LMS` integrates the aforementioned ideas into an efficient method for implementing decomposition. In this algorithm, a method called `MAS-LMS` is iteratively applied until the edge set of the graph gets empty. `MAS-LMS` 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 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 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 GD Algorithm using `Decompose-LMS` for decomposition 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 [5], h is typically a small integer for real-world graphs.

4.2 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, we refer to as RC, is a unique application of random contraction. This methodology has historically been a theoretical tool for addressing cut problems, as noted in [13]. 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) is contracted, the algorithm merges the dictionaries h_u and h_v. To ensure that this merge is done 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 of 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 2 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. 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 RC algorithm is $O(|E| \log(n))$.

Algorithm 2 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'
```

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.

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.

Our contribution in this section is an optimization of the random contraction implementation, drawing inspiration from a graph representation technique presented in [5]. 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. The RC algorithm, utilizing a streamlined data structure, is named RCF.

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 array, "reverse" keeps its counterpart edge (v, u) in a manner that if $value(i) = v$, then $value(reverse(i))$ would rep-

resent 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.

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 store edge weights in a separate array, updating these 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. As 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. Empirical testing on diverse datasets reveals that our array-based implementation RCF consistently outperforms the original algorithm based on dictionaries.

4.3 Early Merging And Splitting

The early merging and splitting algorithm, known as the MSK algorithm, was proposed by Sun et al. [23]. It determines the k-edge connected components 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 exists 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 [5] described in section 4.1. 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 MAS, 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 two subgraphs. The time complexity of the MSK is $O(r \cdot m)$ where r indicates the number of iterations in the MSK algorithm and m is the number of edges in the graph.

Incorrect claim. In the MSK paper, the authors view the algorithm of [5], 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.

The authors of [23] 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 4.1. 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 4.1. 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.

4.4 Local Cut Detection

Chechik et al. in [6] present yet another algorithm for computing maximal k-edge connected components in directed graphs [6], which we refer to as LCD. A directed graph is k-edge connected if it is strongly connected whenever fewer than k edges are removed. While their algorithm targets directed graphs, it can also be used for undirected graphs by replacing each undirected edge with two bidirectional edges.

The main idea of this method is to identify a small subgraph, with at most $\sqrt{m}$ edges, that's well separated 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. 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 run-time of this algorithm that computes k-edge connected components using local cut detection is $(O((2k)^{k+1} \times \sqrt{m} \times m \log n)$, where k represents the edge connectivity. 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, maximum k can be quite large ranging between 30 to 80 for actual graphs. For such magnitudes, the algorithm presented in [6] 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.

5 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 and the source code can be accessed at GitHub [4]. 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. The sizes of these graphs are shown in Table 1. By varying the sizes of these graphs, we can evaluate our algorithms across different graph structures, from small to large.

5.1 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.

[4] `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 1: Datasets

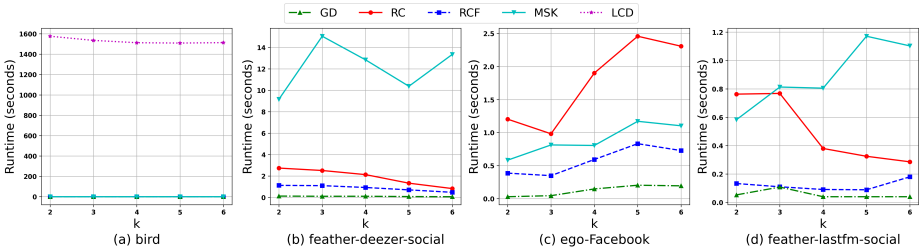

Fig. 2: Performance Analysis on Small Datasets: The figure compares the runtime of various algorithms on small datasets. It underscores the notable efficiency of GD and RCF while illustrating the pronounced slower runtime of LCD.

From Figure 2, 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 2.b, 2.c, and 2.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 2, it can be seen that the GD outperforms other algorithms for small datasets.

5.2 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 3.

In evaluations involving medium and large datasets, the GD algorithm consistently outperforms the other algorithms, much like its performance with smaller

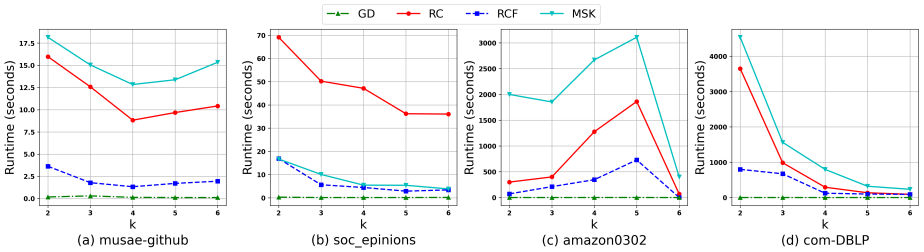

Fig. 3: Performance Analysis on Medium to Large Datasets: The figure shows runtime performance for four algorithms on different graph sizes. It highlights GD's steady efficiency and RCF's advantages over RC. For the largest dataset, amazon, runtime rises with k until $k = 5$ and then sharply drops at $k = 6$. This trend is due to the increase in k-connected components up to $k = 5$, followed by a significant reduction at $k = 6$, as detailed in Table 2.

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, for larger graphs, MSK starts to falter in terms of scalability, a trend that's evident in Figures 3.c and 3.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 3 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 2), 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.

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 Figure 4). We demonstrated the experiment with $k = 2$. However, based on our experiments with other possible values of k, the results followed the same pattern as with $k = 2$.

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 Figure 4, 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. Figure 4 shows 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.

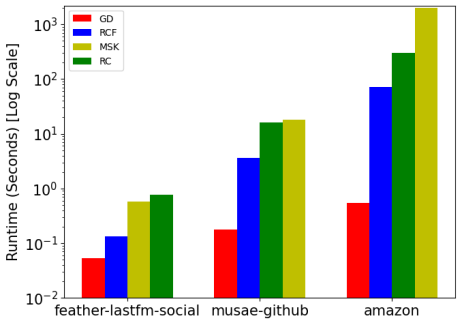

Fig. 4: Comparing running times for GD, RC, RCF, and MSK on three datasets with different sizes for $k = 2$. The figure underscores the consistent efficiency of GD, the waning performance of the RC and MSK on larger datasets, and the subtle transition in the efficiency of RCF from medium to large datasets. A similar trend was observed for $k > 2$ (not shown here).

5.3 Evaluation of Optimization Techniques for RC

In this subsection, we evaluate the effectiveness of the optimization techniques introduced in Section 4.2 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 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, carrying 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 identify these components.

In the plots illustrated in Figure 2 and 3, we executed both the RC and RCF algorithms using the appropriate number of iterations. From the plots, it

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

Table 2: 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, RCF forgoes the forced random contractions that RC does.

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 2, with $k = 4$. The RC algorithm identifies the 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.

6 Discussion

In theory, all the algorithms presented in Section 4 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 $O(m \cdot r)$. 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.

References

1. Akiba, T., Iwata, Y., Yoshida, Y.: Linear-time enumeration of maximal k-edge-connected subgraphs in large networks by random contraction. In: CIKM. pp. 909–918 (2013)
2. Barabási, A.L., Oltvai, Z.N.: Network biology: understanding the cell's functional organization. Nature Reviews Genetics **5**(2), 101–113 (2004)
3. Brin, S., Page, L.: Anatomy of a large-scale hypertextual web search engine. Computer networks and ISDN systems **30**(1-7), 107–117 (1998)
4. Chang, L., Wang, Z.: A near-optimal approach to edge connectivity-based hierarchical graph decomposition. The VLDB Journal pp. 1–23 (2023)
5. Chang, L., Yu, J.X., Qin, L., Lin, X., Liu, C., Liang, W.: Efficiently computing k-edge connected components via graph decomposition. In: SIGMOD. pp. 205–216 (2013)
6. Chechik, S., Hansen, T.D., Italiano, G.F., Loitzenbauer, V., Parotsidis, N.: Faster algorithms for computing maximal 2-connected subgraphs in sparse directed graphs. In: SODA. pp. 1900–1918 (2017)
7. Delling, D., Sanders, P., Schultes, D., Wagner, D.: Engineering route planning algorithms. In: Algorithmics of large and complex networks: Design, analysis, and simulation. pp. 117–139. Springer (2009)
8. Esfahani, F., Daneshmand, M., Srinivasan, V., Thomo, A., Wu, K.: Scalable probabilistic truss decomposition using central limit theorem and h-index. Distributed Parallel Databases **40**(2-3), 299–333 (2022)
9. Esfahani, F., Srinivasan, V., Thomo, A., Wu, K.: Efficient computation of probabilistic core decomposition at web-scale. In: EDBT. pp. 325–336 (2019)
10. Fortunato, S.: Community detection in graphs. Physics reports **486**(3-5), 75–174 (2010)
11. Galil, Z., Italiano, G.F.: Reducing edge connectivity to vertex connectivity. ACM Sigact News **22**(1), 57–61 (1991)
12. Jeong, H., Mason, S.P., Barabási, A.L., Oltvai, Z.N.: Lethality and centrality in protein networks. Nature **411**(6833), 41–42 (2001)
13. Karger, D.R.: Global min-cuts in rnc, and other ramifications of a simple min-cut algorithm. In: SODA. pp. 21–30 (1993)
14. Khaouid, W., Barsky, M., Venkatesh, S., Thomo, A.: K-core decomposition of large networks on a single pc. PVLDB **9**(1), 13–23 (2015)
15. Leskovec, J., Chakrabarti, D., Kleinberg, J., Faloutsos, C., Ghahramani, Z.: Kronecker graphs: An approach to modeling networks. The Journal of Machine Learning Research **11**, 985–1042 (2010)
16. Mislove, A., Marcon, M., Gummadi, K.P., Druschel, P., Bhattacharjee, B.: Measurement and analysis of online social networks. In: IMC'07. pp. 29–42 (2007)
17. Nagamochi, H., Ibaraki, T.: A linear time algorithm for computing 3-edge-connected components in a multigraph. Japan journal of industrial and applied mathematics **9**, 163–180 (1992)
18. Nagamochi, H., Watanabe, T.: Computing k-edge-connected components of a multigraph. IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences **76**(4), 513–517 (1993)
19. Newman, M.: The structure and function of complex networks. SIAM review **45**(2), 167–256 (2003)
20. Page, L., Brin, S., Motwani, R., Winograd, T.: The pagerank citation ranking: Bringing order to the web. In: Stanford InfoLab (1999)

21. Seidman, S.: Network structure and minimum degree. Social networks **5**(3), 269–287 (1983)
22. Spirin, V., Mirny, L.A.: Protein complexes and functional modules in molecular networks. Proceedings of the National Academy of Sciences **100**(21), 12123–12128 (2003)
23. Sun, H., Huang, J., Bai, Y., Zhao, Z., Jia, X., He, F., Li, Y.: Efficient k-edge connected component detection through an early merging and splitting strategy. Knowledge-Based Systems **111**, 63–72 (2016)
24. Tsin, Y.H.: Yet another optimal algorithm for 3-edge-connectivity. Journal of Discrete Algorithms **7**(1), 130–146 (2009)
25. Wang, J., Cheng, J.: Truss decomposition in massive networks. PVLDB **5**(9), 812–823 (2012)
26. Wang, T., Zhang, Y., Chin, F.Y., Ting, H.F., Tsin, Y.H., Poon, S.H.: A simple algorithm for finding all k-edge-connected components. Plos one **10**(9), e0136264 (2015)
27. Wu, J., Goshulak, A., Srinivasan, V., Thomo, A.: K-truss decomposition of large networks on a single consumer-grade machine. In: ASONAM. pp. 873–880 (2018)
28. Yang, J., Leskovec, J.: Defining and evaluating network communities based on ground-truth. Knowledge and Information Systems **42**(1), 181–213 (2015)
29. Zhou, R., Liu, C., Yu, J.X., Liang, W., Chen, B., Li, J.: Finding maximal k-edge-connected subgraphs from a large graph. In: Proceedings of the 15th international conference on extending database technology. pp. 480–491 (2012)