# Linear-Time Enumeration of Maximal *k*-Edge-Connected Subgraphs in Large Networks by Random Contraction

Takuya Akiba
The University of Tokyo
Tokyo, 113-0033, Japan
t.akiba@is.s.u-tokyo.ac.jp

Yoichi Iwata
The University of Tokyo
Tokyo, 113-0033, Japan
y.iwata@is.s.u-tokyo.ac.jp

Yuichi Yoshida
National Institute of Informatics,
Preferred Infrastructure, Inc.
Tokyo, 101-8430, Japan
yyoshida@nii.ac.jp

## ABSTRACT

Capturing sets of closely related vertices from large networks is an essential task in many applications such as social network analysis, bioinformatics, and web link research. Decomposing a graph into *k-core components* is a standard and efficient method for this task, but obtained clusters might not be well-connected. The idea of using *maximal k-edge-connected subgraphs* was recently proposed to address this issue. Although we can obtain better clusters with this idea, the state-of-the-art method is not efficient enough to process large networks with millions of vertices.

In this paper, we propose a new method to decompose a graph into maximal $k$-edge-connected components, based on random contraction of edges. Our method is simple to implement but improves performance drastically. We experimentally show that our method can successfully decompose large networks and it is thousands times faster than the previous method. Also, we theoretically explain why our method is efficient in practice. To see the importance of maximal $k$-edge-connected subgraphs, we also conduct experiments using real-world networks to show that many $k$-core components have small edge-connectivity and they can be decomposed into a lot of maximal $k$-edge-connected subgraphs.

## Categories and Subject Descriptors

G.2.2 [**Discrete Mathematics**]: Graph Theory—*Graph algorithms*

## General Terms

Algorithms, Experimentation, Performance

## Keywords

Graphs, vertex clusters, cohesive subgraphs, connectivity, maximal $k$-edge-connected subgraph, random contraction

*CIKM'13,* Oct. 27–Nov. 1, 2013, San Francisco, CA, USA.
Copyright 2013 ACM 978-1-4503-2263-8/13/10 ...$15.00.
http://dx.doi.org/10.1145/2505515.2505751.

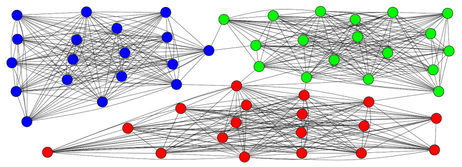


**Figure 1: A $k$-core component in the Arxiv-GrQc dataset that can be decomposed into three maximal $k$-edge-connected subgraphs, where $k = 17$. Some vertices are not shown due to space limit.**

## 1. INTRODUCTION

Sets of closely related vertices from networks, or *clusters*, play an important role in many applications such as social network analysis, computational biology, and web link research. For example, in social networks, cohesive groups can be regarded as communities such as friends, co-workers, neighbors, and so on. In protein-protein interaction networks, they may represent sets of proteins having the same function [13]. In web graphs, they are likely groups of web pages about the same or related topics [17]. Depending on motivations and applications, a myriad of models for clusters were proposed.

Among those, one of the most famous models is $k$-core [25, 14]. In a graph $G = (V, E)$, the *$k$-core* of $G$ is the largest subgraph of $G$ such that all vertices in it have degree at least $k$. Since the $k$-core is not necessarily connected, connected components of the $k$-core, called *$k$-core components*, are often used as clusters. While computing clusters under other models is NP-hard or requires at least $O(|E|^{1.5})$ time, enumerating $k$-core components for all $k$ takes only linear time [5]. This fact and the simplicity of the notion of $k$-core make it prominent in the study of real-world large networks (see Section 2.2 for concrete applications).

However, there is a drawback of adopting $k$-core components as clusters. That is, sometimes a $k$-core component can be easily disconnected. See Figure 1 for an example of a $k$-core component in a real-world network. As is easily seen, the $k$-core component is not well-connected, and it can be divided into three subgraphs by deleting a small number of edges. Thus, it is more natural to regard it consisting of three clusters than a single cluster. The main reason of this issue is that $k$-core only restricts degrees of vertices in subgraphs and it does not use any structure of them. Indeed, as we will discuss in Section 2.1, most of the classical models also have the same risk.

This issue is already known and connectivity is widely used as a measure of cohesion in social network analysis [30, 16]. Algorithmically, Zhou et al. [32] proposed to use *maximal $k$-edge-connected subgraphs* (M$k$ECS) as clusters. An M$k$ECS is a maximal subgraph that remains connected no matter how we remove $k-1$ or fewer edges in it. For example, the $k$-core component in Figure 1 is properly decomposed into three clusters if we adopt M$k$ECSs. It is known that a $k$-core component can be (uniquely) partitioned into M$k$ECSs. Thus, M$k$ECSs can be considered as a refinement of $k$-core components into more tightly connected subgraphs.

Though adopting M$k$ECSs is a promising idea, the algorithm by Zhou et al. [32] is not fast enough and cannot be applied to large networks. The algorithm can be summarized as follows. We first find a cut of size less than $k$ and remove edges in the cut, and then we repeat the same procedure on each connected component in the resulting graph. The most time-consuming part is finding cuts of size less than $k$. Of course, we can find such cuts by using any algorithm that computes the minimum cut. However, a typical algorithm, due to Stoer and Wagner [27], takes $O(|V||E| + |V|^2 \log |V|)$ time and is quite costly. To reduce the number of times to compute the minimum cut, Zhou et al. proposed several heuristics to find small cuts. Though these heuristics are highly effective, still we can only deal with graphs with thousands of vertices. Since real-world networks are far more gigantic, we need new techniques.

## 1.1 Our Contributions

In this paper, we propose a new efficient algorithm to enumerate M$k$ECSs, which is several orders of magnitude faster than the previous method. Now it becomes practical to use M$k$ECSs in place of other models such as $k$-core when analyzing large-scale networks with hundreds of millions of edges and millions of vertices.

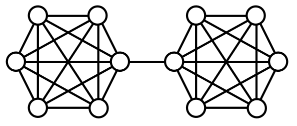
Our algorithm is designed based on a novel application of *random contraction*, which has been a theoretical tool with high affinity for cut problems [19]. Roughly speaking, *contraction* of an edge means removing the edge and merging two endpoints of it. An overview of our method is as follows. Starting with the input graph, as long as there is an edge, we randomly pick an edge and contract it. During the contraction, if we have found a vertex of degree less than $k$, then we cut edges incident to it immediately. We call this process an *iteration.* Note that each (isolated) vertex in the end corresponds to a connected component in the original graph. Then, we repeat the iteration for each connected component a sufficient number of times.

Though simple, our algorithm is efficient since *(i)* each iteration can be implemented to run in linear time, *(ii)* multiple cuts can be found in one iteration, and therefore *(iii)* the number of necessary iterations is small. Moreover, since our method does not rely on complicated combination of heuristics or pruning, it is robust and works well for various types of networks and any choice of $k$.

We explain related works and applications of M$k$ECSs in Section 2. In Section 3, we give definitions used in this paper and review several facts on edge-connectivity. We give an overview and a detailed implementation of our algorithm in Section 4. In Section 5, we show theoretical bounds on the number of necessary iterations. Section 6 is devoted to present our experimental results.

**Table 1: Models of vertex clusters[1].**

| Model | Constraints | Connectivity | Complexity |
|---|---|---|---|
| clique | degree | high | NP-hard |
| quasi-clique | degree | possibly low | NP-hard |
| $k$-plex | degree | possibly low | NP-hard |
| $k$-core | degree | possibly low | $O(\|E\|)$ |
| $r$-clique | distance | possibly low | NP-hard |
| $r$-clan | distance | possibly low | NP-hard |
| $r$-club | distance | possibly low | NP-hard |
| $DN$-Graph [29] | triangles | high | NP-hard |
| $k$-truss [28] | triangles | high | $O(\|E\|^{1.5})$ |
| **M$k$ECS** | **connectivity** | **high** | $O(t\|E\|)$ |


(a) An ill-connected cluster that classical models fail to separate.

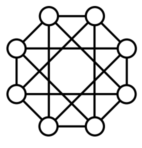

(b) A well-connected cluster that cannot be found with $DN$-Graph and $k$-truss.

**Figure 2: Examples of vertex clusters that other models cannot handle well.**

**Frequently-asked question:** *Is the algorithm heuristic? Is it an approximate method with no guarantee?* — No, absolutely not. First, our algorithm never cuts wrongly. Even if we stop iterations earlier, the complete decomposition is always a refinement of the current decomposition at hand. Second, our algorithm *does* completely decompose a graph into exact M$k$ECSs quickly. We will present a theoretically guaranteed iteration stopping criteria (Section 5.3) and confirm by experiments that the actual number of necessary iterations is so small (Section 6.3.2).

## 2. RELATED WORK

## 2.1 Models of Cohesive Subgraphs

A cluster is often modeled as a subgraph satisfying some conditions, and several notions of clusters have been proposed. We review them and compare with M$k$ECSs.

**Degree-based Models:** A *clique* is a subgraph whose vertex is adjacent to every vertex in it. Since the condition of cliques is too strict, using cliques we often miss important clusters. Therefore, several relaxed notions were proposed. A *$k$-plex* [26] is a maximal subgraph in which every vertex has edges to all but at most $k$ vertices in it. A *quasi-clique* with a parameter $\gamma$ is a subgraph with $n$ vertices and $\gamma\binom{n}{2}$ edges. The problems of enumerating these subgraphs are NP-hard. Moreover, all these models fail to separate subgraphs with high degree and low connectivity. For example, using $k$-plex or quasi-clique, we will obtain the ill-connected subgraph in Figure 2a as one cluster since all the vertices in the subgraph have high degree.

**Distance-based Models:** An *$r$-clique* is a maximal subgraph such that the maximum distance between two vertices in the original graph is at most $r$. An *$r$-clan* is an $r$-clique such that the diameter is also at most $r$. An *$r$-club* is a

[1]Variable $t$ in the time complexity of M$k$ECS is the number of iterations, which is very small as we later show theoretically and empirically.

maximal subgraph such that the diameter is at most $r$. The problems of enumerating these models are also NP-hard. Moreover, all of these models also fail to separate subgraphs with low diameter and low connectivity. For example, using $r$-clique, $r$-clan or $r$-clubs, we will obtain the ill-connected subgraph in Figure 2a as one cluster since the subgraph has small diameter.

**Triangle-based Models:** To address the problems of these classic models, besides M$k$ECSs, notions of a $DN$-graph [29] and a $k$-truss [28] were recently proposed. A subgraph is called a *$DN$-graph* when the minimum number of common neighbors between endpoints over all edges in it satisfies some condition. However, enumerating $DN$-graphs is also an NP-hard problem. Similarly, a *$k$-truss* is the largest subgraph in which every edge is contained in at least $k-2$ triangles. Though $DN$-graphs, $k$-trusses and M$k$ECSs take connectivity into consideration, they are different in nature because a triangle is a very local concept whereas edge-connectivity is a more global concept. In particular, there are a few or no triangles in (almost) bipartite networks such as online dating networks, paper-author networks, product-purchaser networks or movie-actor networks. For these networks, it is better to use M$k$ECSs since we cannot find any cohesive groups with $DN$-graphs or $k$-trusses. For example, the cohesive subgroup in Figure 2b is neither $DN$-graphs nor $k$-trusses for any choice of parameter, since there is no triangle in the subgraph, whereas it is a 4-edge-connected subgraph and can be found by our method.

## 2.2 Applications

Generally, we can use M$k$ECSs in place of any other models we mentioned above. For example, in social networks, M$k$ECSs can be regarded as communities such as friends, coworkers, neighbors, and so on. In protein-protein interaction networks, M$k$ECSs may represent sets of proteins having the same function [13]. In web graphs, M$k$ECSs are likely to be groups of web pages about the same or related topics [17]. Using M$k$ECS, ill-connected subgraphs shown in Figures 1, 2 can be separated properly. In particular, when we want to capture cohesive groups that interact each other well, constraints on connectivity is more direct than constraints on degree, diameter, and triangles. Therefore, M$k$ECS seems to be a more natural model than the classic models.

Especially, as we stated before, M$k$ECSs can be considered naturally as a refinement of $k$-core components. Therefore, we can expect that M$k$ECSs can immediately improve several applications using $k$-core. $k$-core has wide range of applications such as finding the most influential spreaders in social network [20], analyzing structural properties of networks such as hierarchies, self-similarity, and connectivity [2, 3], analyzing and interpreting cooperative process in networks [11], separating and fingerprinting protein complexes in protein-protein interaction networks [1], and capturing clusters in financial activity networks to discover financial crimes [15].

Particularly, in some applications including [2] and [11], $k$-core is used to approximate the connectivity of subgroups. Their justification of using $k$-core is that a $k$-core component is $k$-connected with high probability under the Erdős-Rényi model [23], which is a model of random graphs. It is obvious that M$k$ECSs directly solves their problems. Moreover, experiments in this paper show that actually giant $k$-core components often have small connectivity and split into many subgraphs by small cuts (Section 6.2.1).

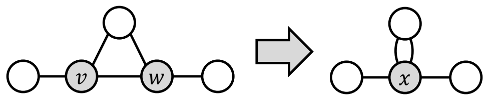


**Figure 3: An example of contraction. Vertex $v$ and $w$ are contracted to a new vertex $x$.**

## 3. PRELIMINARIES

In this paper, we focus on networks that are modeled as undirected and unweighted graphs. Let $G = (V, E)$ be a graph with vertex set $V$ and edge set $E$. We describe the number of vertices $|V|$ as $n$ and the number of edges $|E|$ as $m$. For two disjoint subsets of vertices $S, T \subseteq V$, we define $E(S, T)$ as the set of edges that have one endpoint in $S$ and the other endpoint in $T$. Then, we define $E(S) = E(S, V \setminus S)$, and we write $E(v)$ instead of $E(\{v\})$. Also, we define $d(S, T)$, $d(S)$ and $d(v)$ as the size of $E(S, T)$, $E(S)$ and $E(v)$, respectively. In particular, $d(v)$ is called the *degree* of a vertex $v$. We denote by $G[S]$ the subgraph induced by $S$. For any set of vertices $S \subseteq V$ with $0 < |S| < n$, the edge set $E(S)$ is called a *cut*. A cut with the minimum size is called a *minimum cut*. For notational simplicity, we regard that the minimum size of a cut is $\infty$ when $|V| = 1$.

We formally define contraction of an edge. Let $e = (v, w)$ be an edge in a graph $G = (V, E)$ with $v \neq w$. Let $f$ be a function that maps every vertex in $V \setminus \{v, w\}$ to itself, and otherwise, maps it to a new vertex $x$. The *contraction* of $e$ results in a new graph $G' = (V', E')$, where $V' = V \setminus \{v, w\} \cup \{x\}$ and $E' = \{(f(u), f(v)) \mid (u, v) \in E, f(u) \neq f(v)\}$.

See Figure 3 again for an example of contraction. We note that, if there are edges from $v$ and $w$ to the same vertex, then they will result in parallel edges. However, we never make self-loops by contraction.

In the following two subsections, we review properties of M$k$ECSs and $k$-cores.

## 3.1 Edge Connectivity and Maximal $k$-Edge-Connected Subgraphs

A graph $G$ is called *$k$-edge-connected* if it remains connected no matter how we remove less than $k$ edges. In other words, $G$ is $k$-edge-connected if the minimum size of a cut is at least $k$. From Menger's theorem, it is also equivalent to having at least $k$ edge-disjoint paths between any two distinct vertices. The *edge-connectivity* of a graph $G$ is a maximum $k$ such that $G$ is $k$-edge-connected.

A *maximal $k$-edge-connected subgraph* (M$k$ECS) is a $k$-edge-connected induced subgraph $G[S]$ such that no proper superset $T \supset S$ induces a $k$-edge-connected subgraph. The following property is well-known.

Lemma 3.1. *Let $G = (V, E)$ be a graph and $S_1, S_2 \subseteq V$ be two intersecting vertex sets. If $G[S_1]$ and $G[S_2]$ are $k$-edge-connected, then $G[S_1 \cup S_2]$ is also $k$-edge-connected.*

Corollary 3.1. *Each vertex $v \in V$ belongs to exactly one maximal $k$-edge-connected subgraph.*

Therefore, to enumerate all M$k$ECSs in the graph, it suffices to partition the vertex set $V$ into M$k$ECSs.

**Algorithm 1** Basic Iteration

```
1: procedure ContractAndCut($G$, $k$)
2:     $G' \leftarrow G$
3:     while $G'$ is not empty do
4:         if $\exists u \in V(G')$ such that $d(u) < k$ then
5:             $U \leftarrow$ 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 2** Overall Algorithm

```
1: procedure Decompose($G$, $k$, $t$)
2:     $\mathscr{G}_0 \leftarrow \{G\}$
3:     for $i = 1, 2, \ldots, t$ do
4:         $\mathscr{G}_i \leftarrow \{\}$
5:         for all $G' \in \mathscr{G}_{i-1}$ do
6:             $\mathscr{G}_i \leftarrow \mathscr{G}_i \cup$ ContractAndCut($G'$, $k$)
7:     return $\mathscr{G}_t$
```

## 3.2 *k*-Core Components

Let $G = (V, E)$ be a graph. Let $S \subseteq V$ be the largest set of vertices such that every vertex in $G[S]$ has degree at least $k$. We can easily see such $S$ is uniquely determined and we call the induced subgraph $G[S]$ the *$k$-core* of $G$. We call each connected component in the $k$-core a *$k$-core component*.

Let $S$ be a $k$-core component and $T$ be an M$k$ECS with $|T| \geq 2$, and suppose that $S$ and $T$ intersect. Since every vertex in $G[T]$ has degree at least $k$ from the $k$-edge-connectivity of $G[T]$, every vertex in $G[S \cup T]$ also has degree at least $k$. However, from the maximality of the $k$-core, $T$ must be contained in $S$. Then, a $k$-core component can be partitioned into M$k$ECSs from Corollary 3.1, and M$k$ECSs can be seen as a refinement of $k$-core components.

## 4. ALGORITHM DESCRIPTION

In this section, we first describe a high-level overview of the proposed method (Section 4.1). Next, we discuss how to implement an iteration to run in linear time (Section 4.2). Finally, we introduce the *forced contraction technique*, which drastically reducing the number of necessary iterations by increasing the probability of successfully finding cuts (Section 4.3).

## 4.1 Overview

First, we describe a high-level overview of the proposed algorithm. The idea behind our method is repeatedly finding cuts with size less than $k$ and dividing the graph along these cuts. If we reach the point that each connected component has no cut with size less than $k$, then they are $k$-edge-connected and thus M$k$ECSs (see [32, Theorem 1] for details).

A high-level overview of an iteration of our method is described in Algorithm 1. While there is a vertex with degree less than $k$, we remove it. Otherwise, we choose an edge uniformly at random and contract it. We repeat this process until the graph becomes empty.

As Figure 4 schematically explains, the size of a cut $E(S)$ is equal to the degree of the vertex created by contracting

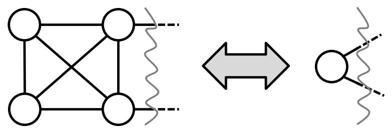


**Figure 4: The relationship between the size of a cut and the degree of the contracted vertex.**

all vertices in $S$. Therefore, the removal of a vertex with degree less than $k$ in the algorithm corresponds to cutting the original graph by removing less than $k$ edges. However, if we have contracted an edge in $E(S)$ before contracting all vertices $S$ into a single vertex, Algorithm 1 fails to find a cut even if a set of vertices satisfies $d(S) < k$. Thus, we repeatedly apply Algorithm 1 with randomization to thoroughly enumerate cuts.

Our overall algorithm is just repeatedly calling Algorithm 1 for all subgraphs we have obtained so far (Algorithm 2). We will see in Section 4.2 that an iteration of our method can be implemented to run in linear time. We will show that the number of iterations $t$ to find all such cuts is indeed small theoretically (Section 5) and experimentally (Section 6.3).

**Comparison with Karger's minimum-cut algorithm:** An algorithm somewhat similar to our method is Karger's minimum-cut algorithm [19]. However, there are significant differences between Karger's algorithm and ours. *(i)* It finds a minimum cut while our method finds cuts of size less than $k$. In particular, our method cannot be achieved just by combining Karger's algorithm and the algorithm by Zhou et al. [32]. Indeed, Karger's algorithm is not a practical minimum-cut algorithm in comparison with other minimum-cut algorithms [12], while our method outperforms the previous method [32], based on a faster minimum-cut algorithm of Stoer and Wagner [27]. *(ii)* Implementation of our method is more involved than Karger's algorithm. Since Karger's algorithm only cares about the number of edges in the final graph, it can skip many computations. However, since we want to find vertices of degree less than $k$ during an iteration, we need several new ideas to obtain a linear-time implementation (Section 4.2). *(iii)* Also, the forced contraction technique is a new significant idea (Section 4.3). It does not change the worst case analysis of Karger's algorithm, but it *does* change the probability we find M$k$ECSs (Section 5).

## 4.2 Linear-time Iteration

Next, we discuss how to efficiently implement the high-level algorithm of Algorithm 1. This is not trivial at all since the algorithm involves contraction of vertices, which is not a standard operation.

Moreover, though our algorithm is similar to Karger's algorithm, efficiently implementing our algorithm is more challenging than implementing Karger's algorithm due to the difference. Karger's algorithm can be easily implemented so that each iteration runs in $O(m)$ time by randomly reordering the edge list first and conducting binary search to determine when the number of vertices in the contracted graph becomes two, instead of fully simulating random contraction. The point is that, in Karger's algorithm, we are interested only in the number of edges between the last two remaining vertices. This idea does not work for our algorithm since we would like to find and remove vertices with degrees less than $k$ during the process.

### 4.2.1 Graph Representation and Contraction

The method we propose is to efficiently simulate random contraction by managing the adjacency list using hash dictionaries and materializing contraction like the weighted quick-find algorithm [31].

Here, we deal with parallel edges by weights. For each vertex $v$, we prepare a hash dictionary $h_v$ that contains edges incident to $v$ as its elements. A key of an edge $(v, w)$ is $w$ and its value is the weight of it.

Instead of randomly selecting an edge each time, we randomly re-order the edge list and process edges in this order. When the endpoints of a selected edge are already contracted to one vertex, we ignore the edge. Otherwise, we contract the endpoints.

To contract an edge $(u, v)$, we have to merge hash dictionaries $h_u$ and $h_v$. We use the following trick to reduce the time complexity. That is, when merging two hash dictionaries $h_u$ and $h_v$, we insert edges from the smaller hash dictionary to the larger one. This can be simply done by inserting all edges of one hash dictionary, say $h_u$, into the other hash dictionary, say $h_v$. Suppose that we are moving an edge to $w$ in $h_u$ to $h_v$. If $h_v$ does not contain the edge $(v, w)$, then we simply add the edge $(v, w)$ to $h_v$ with the same weight as the edge $(u, w)$ in $h_u$. Otherwise, we increase the weight of $(v, w)$ in $h_v$ by the weight of $(u, w)$ in $h_u$. By using this trick, the algorithm speeds up drastically, and we can prove that the worst-case time complexity becomes $O(m \log n)$ and in typical situation the expected time complexity becomes $O(m)$.

### 4.2.2 Worst-Case Time Analysis

We show that our algorithm proposed in Section 4.2.1 runs in $O(m \log n)$ time for any graph and any ordering of edges. To this end, we first discuss a slightly different algorithm and show that its time complexity is $O(m \log n)$. Then, we show that our algorithm is not slower than it.

We consider the following algorithm: let $s(v)$ be the number of vertices in the original graph that are contracted to a vertex $v$. Suppose we want to merge two hash dictionaries $h_u$ and $h_v$. If $s(u) \leq s(v)$, then we insert all edges in the hash dictionary $h_u$ into the hash dictionary $h_v$. Otherwise, we insert all edges in the hash dictionary $h_v$ into the hash dictionary $h_u$.

We count how many times an edge $e$ can be moved to different hash dictionaries. Let $u_0, u_1, \ldots, u_k$ be the sequence of vertices for which the edge $e$ was inserted to their hash dictionaries. Let $s_i$ be the value of $s(u_i)$ when the edge $e$ is inserted to the hash dictionary $h_{u_i}$. Since $s(u_0) = 1$, $s(u_{i+1}) \geq 2s(u_i)$ and $s(u_k) \leq n$, $k$ is at most $\log_2 n$. Since each edge is moved at most $O(\log n)$ times and the number of edges is $O(m)$, the total time complexity is $O(m \log n)$.

Then, we analyze the running time of our algorithm. Note that the only difference of our algorithm and the algorithm described above is how we merge hash dictionaries. Indeed, sequences of graphs obtained in the process are exactly the same. Also, note that the number of moving edges in our algorithm is always smaller than or equal to that in the algorithm described above. Therefore, our algorithm also runs in $O(m \log n)$ time.

### 4.2.3 Expected Time Analysis

Next, we show theoretical evidence that the proposed algorithm works in linear time in practical situations. A crucial observation is that our algorithm can be seen as a variant of the weighted quick-find algorithm for the disjoint set union-find problem [31].

In the *disjoint set union-find problem*, we process a sequence of two types of queries about disjoint sets. One is finding the set containing a specified element, and the other is joining two sets into one set.

The weighted quick-find algorithm manages sets with simple data structures such as arrays or lists, and it joins two sets by inserting all elements in the smaller set into the larger set. It is proved that, on random sequence of (join) queries, the weighted quick-find algorithm runs in expected linear time [21, 9].

Regarding the hash dictionary $h_v$ as a set of neighbors of a vertex $v$, our algorithm works similarly to the quick-find algorithm. The difference is that sets might not be disjoint since different vertices may (usually) have edges to the same vertex. Nonetheless, we can show that our algorithm would run in expected $O(m)$ time.

We prepare $2m$ elements and we regard that each edge has two corresponding elements. Then by merging these elements properly, we can construct a family of $n$ sets that represents neighbors of vertices in the input graph. Then, we run the quick-find algorithm on the family. Though we have constructed the $n$ sets deterministically, the query sequence on the $n$ sets is random. Thus, the expected running time would be $O(m)$ from [9, 21].

Our algorithm can be seen as a variant of the algorithm above, that is, (1) we start with the family of $n$ sets, and (2) we remove elements if they designate the same vertex. These two differences just reduce the time complexity, and it would be also $O(m)$.

## 4.3 Forced Contraction

Finally, we propose a technique that we call the *forced contraction technique*. We will see this technique drastically improves the number of necessary iterations both theoretically (Section 5.2) and experimentally (Section 6.3).

The idea of the forced contraction technique is to immediately contract $u$ and $v$ when the edge between vertices $u$ and $v$ has weight at least $k$. If edge $(u, v)$ has weight at least $k$, then there is no cut with size less than $k$ that separates vertex $u$ and $v$. Therefore, contracting them is safe, in the sense that it never spoil a cut with size less than $k$, and the probability of finding cuts of size less than $k$ becomes higher.

Once a moderate number of vertices $S'$ in an M$k$ECS $S$ are contracted to one vertex, since the number of edges between $S'$ and $S \setminus S'$ is typically high, we can expect that another vertex in $S \setminus S'$ will be contracted to $S'$ by forced contraction. By contracting vertices in $S$ this way, $S'$ will grow to the whole $S$ soon (see Section 6.3.3).

We can easily combine this technique to the implementation given in Section 4.2.1. After contracting two vertices, if the weight of an updated edge becomes at least $k$, then we contract its endpoints in the next step. We continue this process until no edge has weight at least $k$.

## 5. ANALYSIS OF THE NUMBER OF ITERATIONS

In this section, we bound the number of iterations we should perform to completely decompose the input graph into M$k$ECSs. First, instead of directly bound it, we consider the number of iterations to find a cut of size less than $k$ with high probability (say, $\frac{999}{1000}$). Specifically, we show that it can be bounded by $O(s^2)$ without forced contraction (Section 5.1) and $O(\log^2 s)$ with forced contraction (Section 5.2), where $s$ is the size of the smaller side of the cut. Then in Section 5.3, based on these bounds, we argue how to decide in practice the number of iterations throughout our method.

### 5.1 Separating Small M$k$ECSs without Forced Contraction

We bound the number of iterations to find a cut of size less than $k$ without forced contraction. Let $\lambda < k$ be the edge-connectivity of the graph. Then from the submodularity of cuts, there exists a vertex set $S^*$ with $d(S^*) = \lambda$ that is *minimal* with respect to inclusion relation. We note that it suffices to show that the probability $p$ we separate $S^*$ in one iteration is $\Omega(1/s^2)$. Indeed, if this holds, then we separate $S^*$ with high probability in $O(s^2)$ iterations from Markov's inequality.

What we want to compute is the probability that we contract the whole $S^*$ into a single vertex before contracting any edge in $E(S^*)$. The original analysis by Karger [19] bounds the probability that we find some fixed minimum cut. We can use a similar approach here. First note that, since any contraction in $V \setminus S^*$ does not affect the probability, we can assume that $V$ consists of $S^*$ and another vertex $v$. Thus, $|E(S^*, \{v\})| = d(S^*) = \lambda$. Suppose that, in the process of Karger's algorithm, $S^*$ has been contracted to a set of vertices $S'$ of $s'$ vertices without contracting any edge in $E(S)$. Since $G[S^*]$ is $\lambda$-edge-connected from the minimality of $S^*$, the vertex set $S'$ is also $\lambda$-edge-connected. Hence, we have at least $\lambda s'/2$ edges in $S'$. Therefore, the probability that we contract an edge in $E(S')$ in the next step is at most

$$\frac{d(S^*)}{\frac{\lambda s'}{2} + d(S^*)} = \frac{1}{\frac{\lambda s'}{2d(S^*)} + 1} \leq \frac{1}{\frac{\lambda s'}{2\lambda} + 1} = \frac{2}{s' + 2}.$$

Thus, the probability that we find the cut $E(S^*)$ is at least $\prod_{s'=3}^{s} \left(1 - \frac{2}{s'+2}\right) = \frac{12}{(s+2)(s+1)} = \Omega\left(\frac{1}{s^2}\right)$.

### 5.2 Separating Large M$k$ECSs with Forced Contraction

We use the same notations as in the previous section. Now, we turn to analyze the probability $p$ that we separate $S^*$ when using forced contraction, and show that $p$ becomes $\Omega(\frac{1}{\log^2 s})$ from $\Omega(\frac{1}{s^2})$. Using the same argument as before, it implies that we separate $S^*$ in $O(\log^2 s)$ iterations with high probability.

We assume $s \gg \lambda$ since otherwise the analysis in the previous subsection gives a good bound. Similarly to the previous subsection, we can assume that $V = S^* \cup \{v\}$ for a vertex $v$. Since it is hard to analyze the behavior of forced contraction in general, we first assume that $G[S^*]$ forms a complete graph. In the end of this subsection, we show that the same argument applies to random graphs and conclude that $p = \Omega(\frac{1}{\log^2 s})$ holds for typical cases.

We first observe a connection between Karger's algorithm (without forced contraction) on a complete graph and the Erdős-Rényi model [18], which is a model for generating random graphs. In the Erdős-Rényi model, we consider the sequence of graphs $G_0, G_1, \ldots, G_{\binom{n}{2}}$, where $G_0$ is an empty graph of $n$ vertices, and for each $i \geq 1$, we make $G_i$ from $G_{i-1}$ by randomly picking a pair of vertices $(u, v)$ that are not adjacent in $G_{i-1}$ and adding the edge $(u, v)$. In particular, $G_{\binom{n}{2}}$ is exactly $K_n$, where $K_n$ is the complete graph of $n$ vertices.

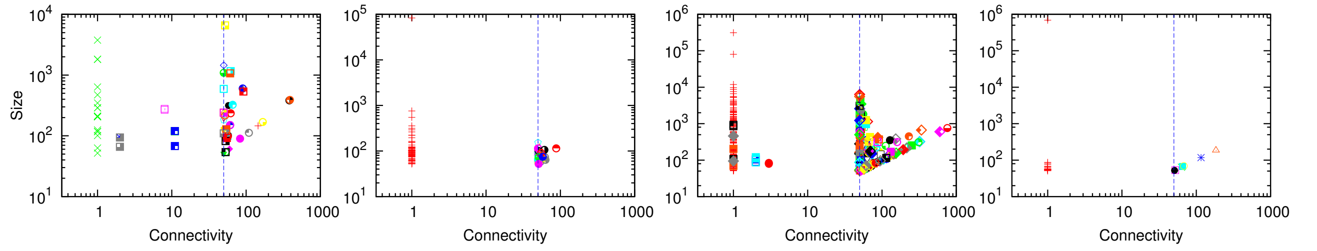
We construct a subsequence $G'_0, \ldots, G'_{n-1}$ of $G_0, \ldots, G_{\binom{n}{2}}$ as follows. We first set $G'_0 = G_0$. Then, we add $G_i$ to the sequence $G'_0, G'_1, \ldots$ when we make $G_i$ from $G_{i-1}$ by adding an edge between two different connected components in $G_{i-1}$. Let $G''_i$ be the graph obtained from $K_n$ by contracting each connected component in $G'_i$. Note that $G''_i$ has exactly $n - i$ vertices, and it is not hard to see that $G''_0, G''_1, \ldots, G''_{n-1}$ is the graph sequence we observe when executing Karger's algorithm on $K_n$. It is known that, in $G_{cn}$ for some constant $c > 1/2$, the largest connected component has size $\Theta(n)$ and the second largest connected component has size $\Theta(\log n)$ with high probability [8]. It means that, for some $i$, $G'_i$ contains a connected component of size $\Theta(n)$, and the number of connected components in $G'_i$ is at least $\Omega(\frac{n}{\log n})$ with high probability. In such a case, $i = n - \Omega(\frac{n}{\log n})$ and $G'_{n-\Omega(\frac{n}{\log n})}$ contains a connected component of size $\Theta(n)$.

Now, we use the connection to analyze the probability $p$ that we separate $S^*$ before contracting any edge in $E(S^*)$. Suppose that we have contracted a constant fraction of vertices $U$ in $S^*$ into one vertex $u$ before contracting any edge in $E(S^*)$. Then, since every vertex in $S^* \setminus U$ has at least $\Theta(s) \gg \lambda$ edges to $u$, the whole $S^*$ will be contracted to one vertex by forced contraction. In the Erdős-Rényi model, this situation corresponds to the case that the current graph $G'_i$ has a connected component of size $\Theta(s)$. From the argument above, this situation happens if we have contracted $S^*$ into $O(\frac{s}{\log s})$ vertices before contracting any edge in $E(S^*)$. Similarly to the analysis in Section 5.1, the probability is at least $\prod_{s'=O(s/\log s)}^{s} \left(1 - \frac{2}{s'+2}\right) = \Omega\left(\frac{1}{\log^2 s}\right)$.

Suppose that $G[S^*]$ is a random graph with $t$ edges. We assume $t = \Omega(s \log s)$ to make sure that $G[S^*]$ is connected with high probability [8]. A crucial observation is that the distribution of $G[S^*]$ is the same as the distribution of $G_t$. Since $G_t$ is connected (with high probability), we can define $G'_0, \ldots, G'_{n-1}$ as above and the same argument follows. In summary, over the choice of $G[S^*]$ and the order of edge contractions, the probability is at least $\Omega\left(\frac{1}{\log^2 s}\right)$.

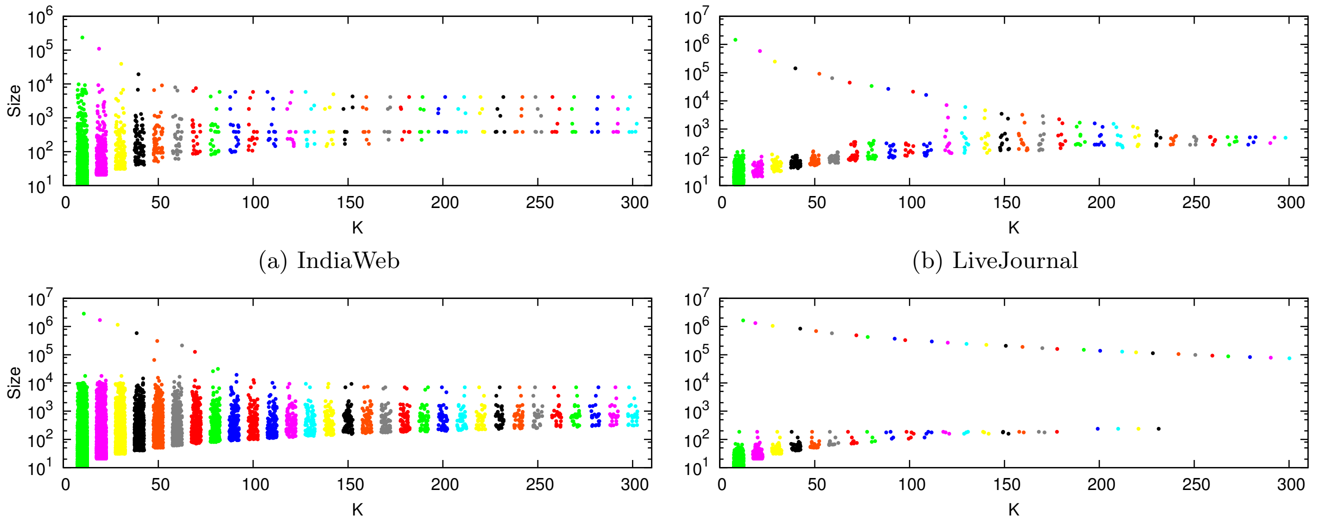
### 5.3 Deciding Number of Iterations in Practice

Now we argue when to stop our method in practice. Recall that we have shown that, with forced contraction, we can find a cut of size less than $k$ with high probability in $O(\log^2 s) = O(\log^2 n)$ iterations. This gives a stopping criteria: if we have not find any new cut of size less than $k$ during $O(\log^2 n)$ iterations, with high probability (say, $\frac{999}{1000}$), the obtained decomposition is the correct decomposition into M$k$ECSs. Thus, we can safely stop our method. In Section 6, by obtaining correct decompositions using the criteria, we experimentally show that about 50 iterations are sufficient in most cases in practice to find all the cuts (Figure 8).



(a) IndiaWeb, $k = 50$ (b) LiveJournal, $k = 50$ (c) IndochinaWeb, $k = 50$ (d) Hollywood, $k = 50$

**Figure 5: The distribution of sizes of M$k$ECSs and edge-connectivity of $k$-core components they belong to. Points with the same color and type denote M$k$ECSs that belong to the same $k$-core component. Blue dashed lines denote $k$.**



(a) IndiaWeb (b) LiveJournal

(c) IndochinaWeb (d) Hollywood

**Figure 6: The distribution of sizes of M$k$ECSs for different $k$'s.**

We should not consider the randomized feature of our method is disadvantage. Rather our method gives a trade-off between the quality of solution and the time complexity. The trade-off is rather good since the number of remaining cuts decreases exponentially, as shown in Section 6 (Figure 7). Thus, it takes a little time to figuring out 99% of cuts, and it would be sufficient for many applications. If we look for a better decomposition, we can keep running our method from the iterative nature of our method. Furthermore, if our method does not find a new cut during $O(\log^2 n)$ iterations, we can safely stop our method by concluding that there is no remaining small cut.

## 6. EXPERIMENTS

The experiments were conducted on a Linux server with Intel Xeon X5670 (2.93 GHz) and 48GB of main memory. The proposed method was implemented in C++ using STL.

### 6.1 Datasets

We conducted experiments on the real-world networks specified in Table 2. Basically we use first two smaller datasets to compare the running time between the proposed method and the previous method, and next four datasets for other experiments. The details of datasets are as follows.

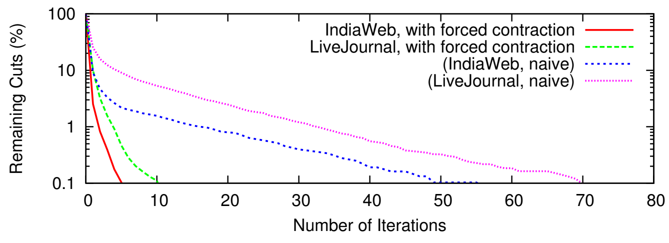
**Table 2: Datasets.**

| Dataset | Vertices | Edges | Type |
|---|---|---|---|
| Arxiv-GrQc | 5,242 | 28,980 | Social Graph |
| Epinions | 75,879 | 405,740 | Social Graph |
| IndiaWeb | 1,382,908 | 16,917,053 | Web Graph |
| LiveJournal | 4,847,571 | 68,993,773 | Social Graph |
| IndochinaWeb | 7,414,866 | 150,984,819 | Web Graph |
| Hollywood | 2,180,759 | 228,985,632 | Social Graph |

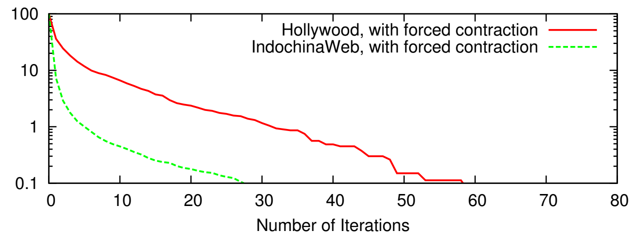
**Arxiv-GrQc** — Arxiv is an on-line archive for preprints of scientific papers (`arxiv.org`). This dataset is the network of collaboration in papers submitted to the general relativity and quantum cosmology category from January 1993 to April 2003, where each node represents an author and each edge represents co-authorship in these papers [22].

**Epinions** — Epinions is an on-line customer review site (`www.epinions.com`). This dataset is the on-line social network in Epinions, where each node represents a user and each edge represents a trust relationship [24].

**IndiaWeb** — This dataset is a web graph of web pages in the `.in` domain, crawled in 2004. Vertices correspond to web pages and edges correspond to hyperlinks [7, 6].

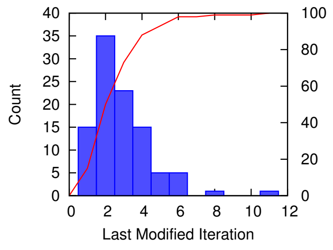


(a) IndiaWeb and LiveJournal, $k = 50$

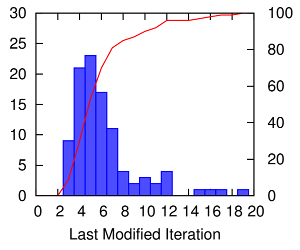


(b) IndochinaWeb and Hollywood, $k = 50$

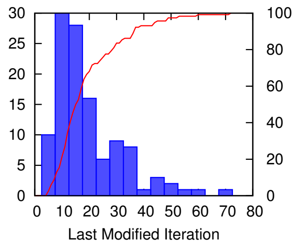
**Figure 7: The number of remaining cuts to be found against the number of iterations.**



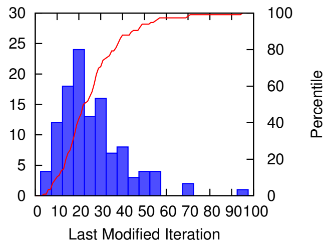
(a) IndiaWeb, $k = 50$



(b) LiveJournal, $k = 50$



(c) Hollywood, $k = 50$



(d) IndochinaWeb, $k = 50$

**Figure 8: Distribution of the number of iterations that the algorithm took to decompose the graph into M$k$ECSs completely. Blue boxes denote the distribution and red lines denote the cumulative distribution.**

**LiveJournal** — LiveJournal is a free on-line social networking website (`www.livejournal.com`). Each vertex corresponds to a member and each edge corresponds to a user-to-user connection [4].

**IndochinaWeb** — This dataset is a web graph of web pages in the country domains of Indochina countries, also crawled in 2004. Vertices correspond to web pages and edges correspond to hyperlinks [7,6].

**Hollywood** — This dataset is a social network of movie actors. Vertices are actors, and two actors are joined by an edge if they appeared in a movie together by 2011 [7,6].

## 6.2 M*k*ECSs in Real-World Networks

First, we investigate properties of M$k$ECSs and $k$-core components in real-world networks to see the importance of M$k$ECS enumeration. In particular, we show these networks contain large $k$-core components that are not tightly connected.

### 6.2.1 *Connectivity of k-Core Components*

We start with evaluating connectivity of $k$-core components to show the importance of computing M$k$ECSs. We computed the distribution of sizes of M$k$ECSs and (estimated) edge-connectivity of $k$-core components that they belong to (Figure 5). Since it is too expensive to compute exact edge-connectivity, we instead use upper bounds on it computed by our method and maximum-flow algorithms.

We can observe that there are a lot of M$k$ECSs that are embedded in $k$-core components with edge-connectivity much lower than $k$. For example, for the IndiaWeb dataset with $k = 50$, there are approximately thirty M$k$ECSs that are in $k$-core components with actual edge-connectivity less than ten. For the LiveJournal dataset with $k = 50$, there is one big $k$-core component with edge-connectivity one that consists of more than sixty M$k$ECSs. Although the number of $k$-core components with edge-connectivity less than $k$ is smaller than the number of $k$-core components with edge-connectivity at least $k$, we can observe that the number of vertices in former $k$-core components are much larger than the number of vertices in latter $k$-core components. Thus, we cannot ignore these $k$-core components just as outliers. Therefore, we conclude that computing M$k$ECSs instead of $k$-core components will lead to better clustering.

### 6.2.2 *Distribution of Subgraph Size*

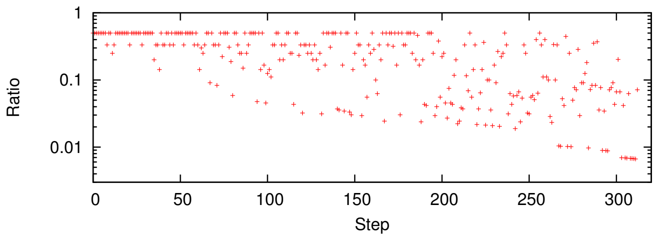
We next show how graphs are decomposed to M$k$ECSs. We computed the distribution of sizes of M$k$ECSs for different $k$'s (Figure 6).

The smaller $k$ is, the more M$k$ECSs we have. Since a subgraph cannot have edge-connectivity larger than its size, when $k$ increases, small subgraphs disappear and only large subgraphs remain. It is also the reason why the minimum size of M$k$ECSs increases when $k$ increases. On the other hand, the maximum size decreases when $k$ increases. This is because these large subgraphs are further decomposed by the stronger requirement on edge-connectivity.

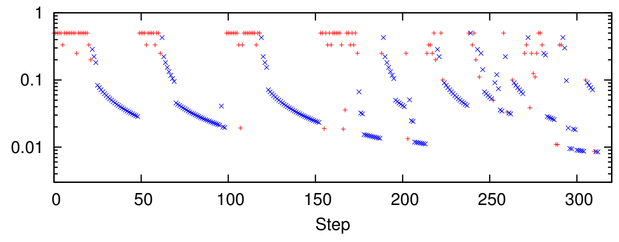
Note that, in every data set, there are one or a few especially huge M$k$ECS(s) and many other smaller M$k$ECSs. This might be explained by the so-called *core-fringe structure* of complex networks [10].

## 6.3 Number of Iterations and Precision

Now, we investigate how our method decomposes a graph as iterations proceed. We analyze the trade-off between the number of iterations and precision, and the number of iterations we need to completely decompose a graph into M$k$ECSs, to show the efficiency of our method and guide users for determining the number of iterations. We also show how the forced contraction technique improves the performance of our method.



(a) Without the forced contraction technique.



(b) With the forced contraction technique.

**Figure 9: Ratio of the numbers of the original vertices that belong to the two vertices contracted at each step. Red points denote random contraction and blue points denote forced contraction.**

**Table 3: Running time (in seconds)**

(a) Comparison with the previous method.

| Dataset | $k$ | Zhou et al. [32] | Proposed | Speedup |
|---|---|---|---|---|
| Arxiv-GrQc | 5 | 374.730 | 0.030 | $1.3 \times 10^4$ |
| | 7 | 5.158 | 0.020 | $2.6 \times 10^2$ |
| | 10 | 0.984 | 0.017 | $5.9 \times 10^1$ |
| Epinions | 10 | $1.283 \times 10^5$ | 1.948 | $6.6 \times 10^4$ |
| | 20 | $2.103 \times 10^4$ | 1.084 | $1.9 \times 10^4$ |
| | 30 | $2.626 \times 10^3$ | 0.871 | $3.0 \times 10^3$ |

(b) Running time for large datasets.

| Dataset | $k$ | | | | | |
|---|---|---|---|---|---|---|
| | 10 | 20 | 40 | 80 | 160 | 320 |
| IndiaWeb | 131 | 67 | 31 | 23 | 17 | 15 |
| LiveJournal | 1,102 | 613 | 215 | 62 | 18 | 9 |
| IndochinaWeb | 6,870 | 5,086 | 2,799 | 921 | 632 | 586 |
| Hollywood | 7,084 | 6,913 | 5,860 | 4,486 | 2,914 | 1,394 |

#### *6.3.1 Number of Remaining Cuts*

First, we investigate the trade-off between number of iterations and precision. We counted the number of remaining cuts to be found against the number of iterations (Figure 7). We ran the algorithm 100 times and took the average for each configuration. Note that the algorithm finally answered the exactly same results for 100 runs.

We can see the number of remaining cuts decreases exponentially. This is because the probability of successfully finding a cut in one iteration is constant, and therefore the probability that we still do not find it decreases exponentially. From this result, we observe that even a small number of iterations results in a meaningful improvement over just outputting $k$-core components.

Also, we can confirm that the forced contraction technique, which we proposed in Section 4.3, is highly effective. For example, for the IndiaWeb dataset and the LiveJournal dataset with $k = 50$, without the forced contraction technique, the algorithm took approximately 50 and 70 iterations to reach 0.1%. However, with the forced contraction technique, it took only 5 and 10 iterations.

#### *6.3.2 Number of Iterations to Complete*

Second, we see how many iterations the algorithm took to completely decompose graphs. We drew the distribution of the number of iterations the algorithm spent to find the last cut (Figure 8). We ran the algorithm 100 times and drew the histogram, but again note that it finally answered the exactly same results for 100 runs. We enabled the forced contraction technique.

On average, the algorithm took only about three iterations for the IndiaWeb dataset with $k = 50$ and six iterations for the LiveJournal dataset with $k = 50$. Almost all executions completely decomposed the graphs within ten iterations, and we never reached twenty iterations. From these results, we conclude that we only need a very small number of iterations to completely decompose graphs.

#### *6.3.3 Ratio of Sizes of Contracted Vertices*

Finally, we see how the forced contraction technique, which we proposed in Section 4.3, changes the behavior of the algorithm. We plotted the ratio of sizes of two vertices contracted at each step in one iteration (Figure 9). More precisely, let $s(v)$ be the number of original vertices that are contracted to vertex $v$. Then, we plotted the value of $\min\{s(v), s(w)\}/(s(v) + s(w))$ where $v$ and $w$ are the vertices contracted at the step. We call $s(v)$ the *size* of a vertex $v$. We used the Arxiv-GrQc dataset and chose $k = 10$.

The first figure, without the forced contraction technique (Figure 9a), looks rather random. The range of ratios widens as the number of steps increases. This is because the maximum size of a vertex increases.

On the other hand, in the figure with the forced contraction (Figure 9b), we can see several blue curves. Each curve represents a process that a vertex that had gained moderate size by random contraction started to absorb nearby vertices one after another by forced contraction. This confirms our explanation in Section 4.3 and the analysis in Section 5.2.

### 6.4 Running Time

Finally, we see the running time of methods to decompose graphs into M$k$ECSs. For the previous method, we used the implementation given by the authors of the method written in Java. To measure time, we ran the program with the same configuration ten times and took the average.

#### *6.4.1 Comparison with the Previous Method*

First, we compare running time of the proposed method and that of the previous method [32] (Table 4a). For the proposed method, we enabled the forced contraction technique and set the number of iterations as ten. We checked that in all executions the two methods output the same result and ten iterations were enough.

We can observe that our method outperforms the previous method by a factor of thousands or even tens of thousands. In particular, the difference is larger for the larger network

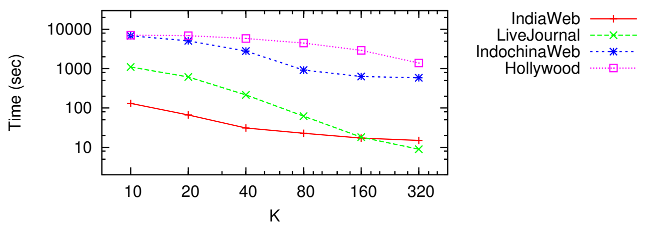


**Figure 10: Running time for large real-world networks against various $k$'s.**

because of the difference of time complexity of algorithms. The difference is also larger for smaller $k$ since the previous method depends on pruning techniques, which are not effective for small $k$. Both the previous method and the proposed method become faster for larger $k$. This is because the size of the $k$-core becomes smaller, and vertices and edges outside the $k$-core are irrelevant.

### 6.4.2 *Performance on Large Networks*

Next, we measured running time for larger real-world networks up to those with hundreds of millions of edges against various $k$'s (Table 4b, Figure 10). We enabled the forced contraction technique. We set number of iterations as twenty for the IndiaWeb dataset and the LiveJournal dataset, and we set it as forty for the IndochinaWeb dataset and the Hollywood dataset. We chose these values since, with probability at least 90%, we can find all cuts as shown in Section 6.3.

It shows that our algorithm has ideal scalability and is quite efficient even for those very large networks. Similarly to the previous section, it takes longer time for smaller $k$ because of the size of the $k$-core.

## 7. ACKNOWLEDGMENTS

Takuya Akiba and Yoichi Iwata are supported by Grant-in-Aid for JSPS Fellows (256563 and 256487). Yuichi Yoshida is supported by JSPS Grant-in-Aid for Research Activity Start-up (24800082), MEXT Grant-in-Aid for Scientific Research on Innovative Areas (24106001), and JST, ERATO, Kawarabayashi Large Graph Project.

## 8. REFERENCES

[1] M. Altaf-Ul-Amin, Y. Shinbo, K. Mihara, K. Kurokawa, and S. Kanaya. Development and implementation of an algorithm for detection of protein complexes in large interaction networks. *BMC Bioinformatics*, 7(1):207, 2006.

[2] J. Alvarez-Hamelin, L. Dall'Asta, A. Barrat, and A. Vespignani. How the $k$-core decomposition helps in understanding the internet topology. In *ISMA Workshop on the Internet Topology*, 2006.

[3] J. I. Alvarez-hamelin, A. Barrat, and A. Vespignani. $k$-core decomposition of internet graphs: hierarchies, self-similarity and measurement biases. *Networks and Heterogeneous Media*, page 371, 2008.

[4] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan. Group formation in large social networks: membership, growth, and evolution. In *SIGKDD*, pages 44–54, 2006.

[5] V. Batagelj and M. Zaversnik. An $o(m)$ algorithm for cores decomposition of networks. *Arxiv preprint cs.DS/0310049*, 2003.

[6] P. Boldi, M. Rosa, M. Santini, and S. Vigna. Layered label propagation: a multiresolution coordinate-free ordering for compressing social networks. In *WWW*, pages 587–596, 2011.

[7] P. Boldi and S. Vigna. The webgraph framework I: compression techniques. In *WWW*, pages 595–602, 2004.

[8] B. Bollobás. *Random graphs*. Cambridge University Press, 2001.

[9] B. Bollobás and I. Simon. On the expected behavior of disjoint set union algorithms. In *STOC*, pages 224–231, 1985.

[10] D. Callaway, M. Newman, S. Strogatz, and D. Watts. Network robustness and fragility: percolation on random graphs. *Phys. Rev. Lett.*, 85(25):5468–5471, 2000.

[11] S. Carmi, S. Havlin, S. Kirkpatrick, Y. Shavitt, and E. Shir. A model of internet topology using $k$-shell decomposition. *PNAS*, 104(27):11150–11154, 2007.

[12] C. S. Chekuri, A. V. Goldberg, D. R. Karger, M. S. Levine, and C. Stein. Experimental study of minimum cut algorithms. In *SODA*, pages 324–333, 1997.

[13] J. Chen and B. Yuan. Detecting functional modules in the yeast protein-protein interaction network. *Bioinformatics*, 22(18):2283–2290, 2006.

[14] J. Cheng, Y. Ke, S. Chu, and M. T. Ozsu. Efficient core decomposition in massive networks. In *ICDE*, pages 51–62, 2011.

[15] W. Didimo, G. Liotta, F. Montecchiani, and P. Palladino. An advanced network visualization system for financial crime detection. In *PacificVis*, pages 203 –210, 2011.

[16] P. Doreian. On the connectivity of social networks. *J. Math. Sociol.*, 3(2):245–258, 1974.

[17] Y. Dourisboure, F. Geraci, and M. Pellegrini. Extraction and classification of dense communities in the web. In *WWW*, pages 461–470, 2007.

[18] P. Erdős and A. Rényi. *On the evolution of random graphs.* Akad. Kiadó, 1960.

[19] D. Karger. Global min-cuts in RNC, and other ramifications of a simple min-out algorithm. In *SODA*, pages 21–30, 1993.

[20] M. Kitsak, L. Gallos, S. Havlin, F. Liljeros, L. Muchnik, H. Stanley, and H. Makse. Identification of influential spreaders in complex networks. *Nature Physics*, 6(11):888–893, 2010.

[21] D. E. Knuth and A. Schönhage. The expected linearity of a simple equivalence algorithm. *Theor. Comput. Sci.*, 6(3):281–315, 1978.

[22] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM Trans. Knowl. Discov. Data*, 1(1), 2007.

[23] T. Luczak. Size and connectivity of the k-core of a random graph. *Discrete Math.*, 91(1):61–68, July 1991.

[24] M. Richardson, R. Agrawal, and P. Domingos. Trust management for the semantic web. In *ISWC*, volume 2870, pages 351–368. 2003.

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

[26] S. Seidman and B. Foster. A graph-theoretic generalization of the clique concept. *J. Math. Sociol.*, 6(1):139–154, 1978.

[27] M. Stoer and F. Wagner. A simple min-cut algorithm. *J. ACM*, 44(4):585–591, 1997.

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

[29] N. Wang, J. Zhang, K.-L. Tan, and A. K. H. Tung. On triangulation-based dense neighborhood graph discovery. *PVLDB*, 4(2):58–68, 2010.

[30] D. White and F. Harary. The cohesiveness of blocks in social networks: Node connectivity and conditional density. *Sociol. Methodol.*, 31(1):305–359, 2001.

[31] A. C.-c. Yao. On the average behavior of set merging algorithms. In *STOC*, pages 192–195, 1976.

[32] 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.