

Available from: http://doi.acm.org/10.1145/2247596.2247652


This is the author's version of the work, posted here with the permission of the publisher for your personal use. No further distribution is permitted. You may also be able to access the published version from your library. The definitive version is available at http://dl.acm.org/.
Finding Maximal k-Edge-Connected Subgraphs from a Large Graph

Rui Zhou, Swinburne University of Technology, Melbourne, Australia; rzhou@swin.edu.au
Chengfei Liu, Swinburne University of Technology, Melbourne, Australia; cliu@swin.edu.au
Jeffrey Xu Yu, The Chinese University of Hong Kong, Hong Kong, China; yu@se.cuhk.edu.hk
Weifa Liang, School of Computer Science, Australian National University, Canberra, Australia; wliang@cs.anu.edu.au
Baichen Chen, School of Computer Science, Australian National University, Canberra, Australia; baichen@cs.anu.edu.au
Jianxin Li, Swinburne University of Technology, Melbourne, Australia; jianxinli@swin.edu.au

ABSTRACT

In this paper, we study how to find maximal k-edge-connected subgraphs from a large graph. k-edge-connected subgraphs can be used to capture closely related vertices, and finding such vertex clusters is interesting in many applications, e.g., social network analysis, bioinformatics, web link research. Compared with other explicit structures for modeling vertex clusters, such as quasi-clique, k-core, which only set the requirement on vertex degrees, k-edge-connected subgraph further requires high connectivity within a subgraph (a stronger requirement), and hence defines a more closely related vertex cluster.

To find maximal k-edge-connected subgraphs from a graph, a basic approach is to repeatedly apply minimum cut algorithm to the connected components of the input graph until all connected components are k-connected. However, the basic approach is very expensive if the input graph is large. To tackle the problem, we propose three major techniques: vertex reduction, edge reduction and cut pruning. These speed-up techniques are applied on top of the basic approach. We conduct extensive experiments and show that the speed-up techniques are very effective.

1. INTRODUCTION

Graphs are used to express the relationships of different objects for a wide range of applications. In social network analysis, individuals can be represented by vertices, and their friendship relations can be represented by edges. In bioinformatics, graphs can be used to model protein interactions and gene coexpressions. In web data management, web pages and their links can be considered as vertices and edges respectively. The common theme of these modelings is to represent an entity as a vertex (or node), and the relationship between two entities as an edge. As a result, many real-life problems can be transformed into mathematical problems on a graph, and then can be tackled with elegant solutions on the shelf.

In graph theory, connectivity is a fundamental subject. It has applications in a variety of traditional areas, such as network reliability analysis [8], VLSI chip design [14], transportation planning [3]. A k-edge-connected graph is a connected graph that cannot be disconnected by removing less than k edges, similarly, a k-vertex-connected graph is a connected graph that cannot be disconnected by removing less than k vertices. We only focus on edge connectivity in this paper, because k-vertex-connectivity can be reduced to k-edge-connectivity, so k-connected is short for k-edge-connected from now on.

On new types of data, finding k-connected subgraphs may be interesting as well. For example, in social network analysis, a k-connected subgraph could approximately model a community, here, k can be defined by a user to express how close the relationships are between members within a community. Different users may be interested in different k’s. Efficiently discovering k-connected subgraphs helps users identify those closely related individuals, and such information could be useful for social behavior mining [2], viral marketing [4], etc. In computational biology, a k-connected subgraph could model a set of genes within the same functional module [26], here vertices represent the genes and edges represent coexpression relationships between the genes. A high-connected subgraph from a gene coexpression graph is likely to capture a functional gene cluster. Finding such subgraphs may assist biologists to analyze gene microarrays and develop reasonable conjectures before experiments. For a web-link graph, a high-connected subgraph may be a collection of web pages talking about a certain topic or discussing related topics. Such subgraphs may be useful for entity association mining from web pages or building a knowledge database based on web pages.

In a word, a k-connected subgraph captures a vertex cluster, where vertices within the cluster are closely related. There are some other defined structures playing a similar role, e.g., clique, quasi-clique (defined on vertices [30] or edges [1]), k-core [24], k-plex [23], etc. A clique defines a
structure where every vertex is connected to the other vertices within the structure. Quasi-clique is a relaxed form of clique, and it asks a vertex to connect with other vertices no less than a predefined percentage, e.g., in a \( n \)-vertex \( \gamma \)-quasi-clique, each vertex is connected to at least \( \lceil \gamma \cdot (n - 1) \rceil \) other vertices. In a \( k \)-core, each vertex is connected to at least \( k \) other vertices. Similarly, in a \( n \)-vertex \( k \)-plex, each vertex is connected to at least \( n - k \) vertices. After all, in presence of these existing explicit structures, why do we need to study \( k \)-connected subgraphs? We explain it in the next paragraph.

Firstly, cliques are too strong, because, in many real scenarios, it is unlikely that every entity would have a link to every other entity within the cluster. On the other hand, quasi-clique, \( k \)-core, \( k \)-plex are sort of weak in some situations. For example, in Fig. 1 (a), an 8-vertex graph is a 3/7-quasi-clique (defined on vertices), because each vertex is connected to at least three of the other vertices in the graph. Fig. 1 (b) is also a 3/7-quasi-clique. Comparing Fig 1 (a) and Fig 1 (b), they are both 3/7-quasi-cliques, having the same number of vertices and edges, and the same degree on each vertex. However, it is more appropriate to say: Fig. 1 (a) contains one vertex cluster while Fig. 1 (b) contains two vertex clusters. In Fig. 1 (c), the whole graph is a 5-core, because each vertex is connected to at least five other vertices. Its subgraph \{A,B,C,D,E,F\} (in a dashed rectangle) is also a 5-core. Comparing Fig 1 (c) and Fig 1 (b), they both 5-cores, but Fig. 1 (c) should be considered as two vertex clusters. \( k \)-plex is similar to \( k \)-core, and has a similar problem.

The above discussion reminds us that connectivity in a subgraph is not negligible. Unfortunately; most existing defined structures are based on node degrees, ignoring the connectivity within the defined subgraph. It is well-known that checking connectivity is more expensive than checking node degrees. As a result, an efficient approach to discover \( k \)-connected subgraph is highly sought after. In this paper, we aim to find all maximal \( k \)-connected subgraphs, that is, \( k \)-connected subgraphs not contained in other \( k \)-connected subgraphs (a formal definition will be given in Section 2), otherwise (if not maximal) we can find too many \( k \)-connected subgraphs.

To guarantee the resulting subgraphs are \( k \)-connected, cut-based processing steps are unavoidable. A basic approach is to repeatedly run a minimum cut algorithm on the connected components of the input graph, and decompose the connected components if a less-than-\( k \) cut can be found, until all connected components are \( k \)-connected. Such solution is acceptable on smaller graphs, but is very expensive on large graphs. To tackle the problem, we design a set of speed-up methods. On one hand, we try to reduce the size of graph so that any cut algorithm can run faster on smaller graphs. This includes vertex reduction and edge reduction. On the other hand, we introduce some pruning conditions with which we can tell directly whether a connected component is \( k \)-connected or not.

We summarize our contributions as follows:

- We show that \( k \)-connected subgraphs may be a better means to model node clusters, compared with some existing models, such as \( k \)-cores and quasi-cliques.

- We propose a basic minimum-cut-based approach to find maximal \( k \)-connected subgraphs. More importantly, we propose three speed-up methods, node reduction, edge reduction and cut pruning that can dramatically improve the performance of the basic approach. The correctness of the speed-up methods are proved theoretically.

- We conduct extensive experiments to test the algorithm performance when applying node reduction, edge reduction and cut pruning on top of the basic approach. The experiment results confirm that the speed-up methods are very effective.

Here is a roadmap of this paper. In Section 2, we provide a formal definition of the problem, and introduce some necessary notations. In Section 4, we introduce vertex reduction and show its correctness. Edge reduction is introduced in Section 5. In Section 6, we introduce how to avoid cutting a connected component and how to cut a connected component into two halves earlier. Experiment results are shown in Section 7. Related works and conclusions are in Section 8 and Section 9 respectively.

2. PRELIMINARIES

We model networked data as a simple, unweighted, undirected graph \( G = (V, E) \), where \( V \) is a set of vertices and \( E \) is a set of edges between vertices. Normally, \( V \) represents entities and \( E \) represents the relationships between entities. In real world, there may be different types of relationships between two entities, but in this paper, we do not distinguish the types of relationships. That means, as long as two entities are related, no matter how many types of relationships there are, we consider the two entities are connected by a single edge.

A graph \( G \) is \( k \)-connected (short for \( k \)-edge-connected) if the removal of any up to \( k - 1 \) edges does not make \( G \) disconnected, and there exists an edge set \( E_{\text{cut}} \) with \( |E_{\text{cut}}| = k \) whose removal will make \( G \) disconnected. The edge set \( E_{\text{cut}} \) is called the cutset of a minimum cut. Note that \( G \) may have more than one minimum cut.

Figure 1: Compare \( k \)-connected subgraph with other structures
A graph $G_s = (V_s, E_s)$ is called an induced subgraph of a graph $G = (V, E)$, when $V_s \subseteq V$, $E_s \subseteq (V_s \times V_s) \cap E$, and for any two vertices $x, y \in V_s$, edge $(x, y) \in E_s$ if and only if $(x, y) \in E$. Usually, an induced subgraph with vertex set $V_s$ from $G$ is denoted as $G[V_s]$.

A subgraph $G_s = (V_s, E_s)$ is a maximal $k$-connected subgraph of $G$, if there does not exist another $k$-connected subgraph $G_s' = (V_s', E_s')$, such that $V_s \subseteq V_s'$ and $E_s \subseteq E_s'$. Apparently, it implies that a maximal $k$-connected subgraph is an induced subgraph.

The problem we will study in this paper is, given a graph $G$ and a user-specified integer $k$, how to find all maximal $k$-connected subgraphs from $G$ efficiently.

3. BASIC APPROACH

In this section, we give a basic approach to find all maximal $k$-connected subgraphs. Important speed-up techniques will be introduced in Section 4, 5 and 6.

The idea of the basic approach is to repeatedly apply any minimum cut algorithm to the graph until each connected component is either a single vertex or a $k$-connected subgraph. Algorithm 1 describes the process. Throughout the process, $R_0$ stores the intermediate results for the graph decomposition, i.e. the produced connected components. If a produced connected component $G_1$ is at least $k$-connected, it will be added into the result set $R$ (line 8); otherwise, $G_1$ will be decomposed into two pieces $\{G_2, G_3\}$ and added into $R_0$ for later inspection (line 5-6). Theorem 1 guarantees the correctness of the algorithm.

**Theorem 1.** Given a graph $G$ and a connectivity threshold $k$, Algorithm 1 correctly finds all maximal $k$-connected subgraphs from $G$.

**Proof.** Obviously, all subgraphs in the result set $R$ are $k$-connected. We need to show each of them is maximal as well. Suppose a graph $G_0 = (V_0, E_0) \in R$ is not maximal, then there must be a maximal $k$-connected subgraph $G_{max} = (V_{max}, E_{max})$ such that $V_0 \subset V_{max}$, and there must also exist a cut in a certain loop produced by the step 3, which separates a vertex (or some vertices) in $V_{max}$ away from $V_0$. However, such a cut cannot exist, because $G[V_{max}]$ is supposed to be $k$-connected. As a result, $G_0$ should be maximal.

To show Algorithm 1 has found “all” maximal $k$-connected subgraphs: let $(v_1, v_2)$ be an edge in a $k$-connected subgraph, since $V_1$ and $V_2$ are $k$-connected, $(v_1, v_2)$ cannot be removed in the first loop (line 3-9) in Algorithm 1. Similarly, $(v_1, v_2)$ cannot be removed in later loops. As a result, $(v_1, v_2)$ will not be removed by Algorithm 1. This completes the proof of the “all” part. The theorem thus is correct.

In Algorithm 1, the critical step is Step 3, i.e. performing a cut-based algorithm on a graph. In fact, it is likely that the cut-based algorithm cannot be avoided because $k$-connectivity needs to be guaranteed on the resulting subgraphs. As a result, if we can speed up the cut-step (Step 3), the $k$-connected subgraph discovery process can be accelerated. It is obvious that a fast minimum cut algorithm is preferred for Step 3. However, in this paper, most of the time, we constrain ourselves to a general minimum cut algorithm, because we aim to design a framework to accommodate any minimum cut algorithm, not a particular one. As such, if a novel minimum cut algorithm would be found, it could then be plugged into our framework without any modification. In case, users feel overwhelmed by the number of minimum cut algorithms to choose from, we suggest one minimum cut algorithm and explain the reason in Section 6. Given a general minimum cut algorithm, we briefly discuss some ideas to accelerate the cut algorithm on graph $G_1$, the details will be unfolded in Section 4, 5, and 6.

- Reduce the size of $G_1$: The performance of most minimum cut algorithms are affected by the size of the graph, i.e. the number of vertices and the number of edges. Therefore, it is desirable if we can safely reduce the size of $G_1$ without affecting its connectivity, or exactly speaking without affecting the $k$-connectivity of those maximal $k$-connected subgraphs of $G_1$. Consequently, we can run a cut algorithm on a smaller graph but produce the same result. Vertex reduction and edge reduction will be introduced in Section 4 and Section 5 respectively.

- Avoid performing the cut algorithm: Some readers may have noticed that an unpromising connected component (with no $k$-connected subgraph inside) may be found earlier, no need to be cut into a few single vertices. For example, if a simple graph $G_1 = (V_1, E_1)$ has no more than $k$ vertices ($|V_1| \leq k$), $G_1$ is at most $(k - 1)$-connected (when $G_1$ is a clique), and cannot be $k$-connected. So $G_1$ can be disregarded earlier. Such speed-up tricks will be elaborated in Section 6.

4. VERTEX REDUCTION

In this section, we aim at reducing the number of vertices to speed up the basic minimum cut operation so that the whole discovery process can be accelerated. The idea is that if a subgraph $G_s$ is $k$-connected we can safely contract the subgraph $G_s$ into a new vertex $v_{new}$ and the size of the original graph is reduced accordingly.

4.1 Contracting a $k$-connected subgraph

We will introduce the contraction process first and explain why this procedure is safe afterwards. Assume we have got a $k$-connected subgraph $G_s = (V_s, E_s)$, the contraction of $G_s$ is as follows: (1) all vertices in $V_s$ are replaced with a new vertex $v_{new}$; (2) all edges between vertices belonging to $V_s$...
contraction, there will be two edges between

though the original graph is simple, e.g., there are two edges

we have

will be disregarded if

Suppose there exists a cut $E_{\text{cut}}$ ($|E_{\text{cut}}| < k$) of $G$ separating

with $v_1$ and $v_2$ into two components $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$

with $v_1 \in V_1$, $v_2 \in V_2$, (1) if $\{\bar{v}_1, ..., \bar{v}_3\} \subseteq V_1$, then $|E_{\text{cut}}| < k$

contradicts that there are $k$ distinct paths from $\{\bar{v}_1, ..., \bar{v}_3\}$ to $v_2$; (2) otherwise, let $\bar{v}_i$ be a vertex separated into $V_2$ by $E_{\text{cut}}$, given $G_s$ is $k$-connected and $v_1, \bar{v}_i \in V_s$, we have $v_1$ and $\bar{v}_i$ are $k$-connected, which also contradicts the existence of

Theorem 2. Given a graph $G = (V, E)$, let $G_s = (V_s, E_s)$ be a $k$-connected subgraph of $G$, let $G' = (V', E')$ be the graph produced from $G$ by contracting $G_s$ into a vertex $v_{\text{new}}$,

for any vertex $v \in V$, we define $\text{image}(v) \in V'$ as: (1) $\text{image}(v) = v_{\text{new}}$, if $v \in V_s$; (2) $\text{image}(v) = v$, if $v \in V \setminus V_s$ (remains the same), then we have: for any vertices $v_1, v_2 \in V$, $v_1, v_2$ are $k$-connected in $G$, if and only if $\text{image}(v_1) = \text{image}(v_2) = v_{\text{new}}$ or $\text{image}(v_1)$ and $\text{image}(v_2)$ are $k$-connected in $G'$.

Proof. proof of “only if”, given $v_1, v_2$ are $k$-connected in $G$:

Case (1): $v_1, v_2 \in V_s$, then obviously we have $\text{image}(v_1) = \text{image}(v_2) = v_{\text{new}}$.

Case (2): Without loss of generality, let $v_1, v_2 \in V \setminus V_s$, i.e. $\text{image}(v_1) = v_{\text{new}}$, $\text{image}(v_2) = v_2$, since $v_1, v_2$ are $k$-connected in $G$, there are $k$ distinct paths between $v_1$ and $v_2$ in $G$, denoted as $\{p_1, ..., p_k\}$. Given one of these paths $p_i$, let $v_{\text{last}}$ be the last vertex on $p_i$ from $v_1$ to $v_2$ satisfying $\text{image}(v_{\text{last}}) = v_{\text{new}}$ and $p_i' = p_i_{\text{last}} \rightarrow v_{\text{last}}$ denote the segment of $p_i$ from $v_{\text{last}}$ to $v_2$ in $G$, it is not difficult to see that $p_i'$ is also a path from $v_{\text{new}}$ to $v_2$ in $G'$. Given that $p_i'$ is a part of $p_i$, together with that $\{p_1, ..., p_k\}$ are distinct paths, we have $\{p_1, ..., p_k\}$ are distinct paths from $v_{\text{new}}$ to $v_2$ in $G'$.

As a result, $v_{\text{new}}$, $v_2$ (or equally $\text{image}(v_1)$, $\text{image}(v_2)$) are $k$-connected in $G'$.

Case (3): $v_1, v_2 \in V \setminus V_s$, i.e. $\text{image}(v_1) = v_1$, $\text{image}(v_2) = v_2$. Since $v_1$ and $v_2$ are $k$-connected in $G$, again there are $k$ distinct paths between them in $G$, denoted as $\{p_1, ..., p_k\}$. Given one of these paths $p_i$, let $v_{\text{first}}$ be the first and last vertices on $p_i$ from $v_1$ to $v_2$ satisfying $\text{image}(v_{\text{first}}) = v_{\text{new}}$ and $\text{image}(v_{\text{last}}) = v_{\text{new}}$ (here, $v_{\text{first}}$, $v_{\text{last}}$ may be the same vertex.), let $p_{i_{\text{first}}} \rightarrow v_{\text{first}}$ be a segment of $p_i$ from $v_1$ to $v_{\text{first}}$ in $G$, $p_i_{\text{first}} \rightarrow v_{\text{last}}$ be a segment of $p_i$ from $v_{\text{last}}$ to $v_2$ in $G$, we can obtain $p_i = p_{i_{\text{first}}} \rightarrow v_{\text{first}} + p_{i_{\text{last}}} \rightarrow v_{\text{last}}$ by concatenating the two segments $p_{i_{\text{first}}} \rightarrow v_{\text{first}}$ and $p_{i_{\text{last}}} \rightarrow v_{\text{last}}$ at $v_{\text{first}}$ and $v_{\text{last}}$. It is not difficult to see that $p_i'$ is a path from $v_1$ to $v_2$ in $G'$. Similarly, since $\{p_1, ..., p_k\}$ are distinct paths, we have $\{p_1', ..., p_k'\}$ are distinct paths in $G'$. As a result, $v_1, v_2$ (or equally $\text{image}(v_1)$, $\text{image}(v_2)$) are $k$-connected in $G'$.

Note that, for a path $p$, $v_{\text{first}}$, $v_{\text{last}}$ may not always exist. If so, let $p_i' = p_i$ and the other parts of the proof remain the same.

Proof of “if”, given either image($v_1$) = image($v_2$) = v_{\text{new}} or image($v_1$) and image($v_2$) are k-connected in G'.

Case (1): image($v_1$) = image($v_2$) = v_{\text{new}}, then obviously $v_1, v_2 \in V_s$. As $G_s$ is a $k$-connected subgraph, $v_1$, $v_2$ is k-connected in $G$.

Case (2): Without loss of generality, let $v_1 \in V_s$, $v_2 \in V \setminus V_s$, i.e. image($v_1$) = v_{\text{new}}, image($v_2$) = v_2, since v_{\text{new}} and v_2 are k-connected in G', there are k distinct paths from v_{\text{new}} to v_2 in G'. Recall that v_{\text{new}} in G' actually represents multiple vertices in G, as a result, there are k distinct paths from subgraph G_s to v_2. We denote the starting vertices of these k distinct paths as $\{v_1, ..., v_k\}$, here $\{v_1, ..., v_k\} \subseteq V_s$ and image($v_1$) = ... = image($v_k$) = v_{\text{new}}. Note that $v_i$, $v_j$ (i ∈ [1, k], j ∈ [1, k], i ≠ j) may be the same vertex. Now we will prove $v_1$, $v_2$ is k-connected in G by contradiction. Suppose there exists a cut $E_{\text{cut}}$ ($|E_{\text{cut}}| < k$) of G separating

and last vertices on $G_s$ into two segments $G_1$ and $G_2$, respectively.

v_1$ to $v_2$ by $E_{\text{cut}}$, given $G_s$ is k-connected and $v_1, \bar{v}_i \in V_s$, we have $v_1$ and $\bar{v}_i$ are k-connected, which also contradicts the existence of

Lemma 1. Given a graph $G = (V, E)$ and three vertices $\{v_a, v_b, v_c\} \subseteq V$, if $v_a, v_b$ are k-connected and $v_a, v_c$ are k-connected, then $v_a, v_c$ are k-connected.

Proof. We prove the lemma by contradiction. Suppose $v_a, v_c$ are not k-connected, there must exist a cut $E_{\text{cut}}$ separating $G$ into two components $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$ with $|E_{\text{cut}}| < k$. Without lose of generality, assume $v_a \in V_1$ and $v_c \in V_2$, since $V_1 \cup V_2$ and $V_1, V_2$ are disjoint, we must be in either $V_1$ or $V_2$. If $v_a \in V_1$, the existence of $E_{\text{cut}}$ contradicts that $v_a, v_c$ are k-connected, otherwise the existence of $E_{\text{cut}}$ contradicts that $v_a, v_c$ are k-connected.

With Theorem 2, we can safely proceed to deal with the contracted graph, but now the problem is how to discover a few k-connected subgraphs in advance.

4.2 Finding a few k-connected subgraphs (not necessarily maximal)

We propose three methods to initially discover a few k-connected subgraphs. Intuitively, the more and the larger these discovered k-connected subgraphs are, the better the reduction effect can be achieved. However, it may take more time to discover more and larger k-connected subgraphs, and thus degrade the overall performance. It is unlikely to know a reasonable trade-off between the two aspects in advance. In our design, we put method efficiency at the first place, and the size of the initially discovered subgraphs at second due to the following reasons:

- It is difficult to find a maximal k-connected subgraph
by expanding an existing $k$-connected subgraph$^1$. We will give an example to show this in section 4.2.3. Consequently, it may not be worth the effort to find as many and as large $k$-connected subgraphs as possible, since finding these temporary subgraphs does not provide a shortcut to finding maximal $k$-connected subgraphs.

- To find $k$-connected subgraphs is only a subprocedure to reduce the number of vertices. It does not need to be perfect, but needs to be fast. So fast methods with reasonable quality are sufficient.

4.2.1 Using materialized views

If there are some precomputed maximal $k'$-connected subgraphs, as either materialized views or historical query results, we may use them as the bases to explore a few $k$-connected subgraphs.

- Case 1: A maximal $k'$-connected subgraph $G'$ has $k' \geq k$, obviously $G'$ is also $k$-connected, but may not be maximal at $k$. If we have all maximal $k'$-connected subgraphs ($k' > k$), then we can safely contract these $k'$-connected subgraphs into a few supernodes (points) by Theorem 2. The size of the resulting graph is then significantly reduced in comparison with the original graph. To make the contraction more effective, we can first expand those materialized $k'$-connected subgraphs to obtain a set of larger $k$-connected subgraphs using the technique in Section 4.2.3, and then contract these $k$-connected subgraphs.

- Case 2: If a maximal $k'$-connected subgraph $G'$ has $k' < k$, then $G'$ may contain induced subgraphs which are $k$-connected. In such case, if $G' = (V', E')$ is not very large (e.g., $|V'| + |E'| \leq B$, where $B$ is a predefined bound), we can find all maximal $k$-connected subgraphs from $G'$ directly; otherwise, further vertex reduction and edge reduction can be performed on $G'$. Note that if we have got all maximal $k'$-connected subgraphs (when $k' < k$), we can start from these $k'$-connected subgraphs without resorting to the original graph, because a $k$-connected subgraph is also $k'$-connected and must be subsumed in one of those maximal $k'$-connected subgraphs (Lemma 2).

**Lemma 2.** For a given graph, its maximal $k$-connected subgraphs are disjoint, i.e. If $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$ are two maximal $k$-connected subgraphs of the same graph $G$ and $G_1 \neq G_2$, we have $V_1 \cap V_2 = \emptyset$.

**Proof.** We prove the lemma by contradiction. Suppose $V_1 \cap V_2 \neq \emptyset$ and let $v \in V_1 \cap V_2$, construct a new induced graph $G[V_1 \cup V_2]$, then for any two vertices $v_1, v_2 \in V_1 \cup V_2$, if $v_1, v_2 \in V_1$ or $v_1, v_2 \in V_2$, obviously $v_1, v_2$ are $k$-connected in $G[V_1 \cup V_2]$ since $G_1, G_2$ are $k$-connected and $G_1, G_2$ are subgraphs of $G[V_1 \cup V_2]$; otherwise, without loss of generality, let $v_1 \in V_1, v_2 \in V_2$, from $v_1, v \in V_1$, we have $v_1, v$ are $k$-connected in $G[V_1 \cup V_2]$, similarly $v_2, v$ are also $k$-connected in $G[V_1 \cup V_2]$. According to Lemma 1, $v_1, v_2$ are $k$-connected in $G[V_1 \cup V_2]$. As a result, $G[V_1 \cup V_2]$ is $k$-connected, but this contradicts that $G_1, G_2$ are maximal $k$-connected subgraphs. Consequently, we have $V_1 \cap V_2 = \emptyset$. $\square$

In summary, as long as there is a precomputed maximal $k'$-connected subgraph $G'$ (no matter $k'$ is larger or smaller than $k$), we can use $G'$ to help discover maximal $k$-connected subgraphs.

4.2.2 Using vertices with high degrees

The second method to find a few $k$-connected subgraphs is a heuristic method. It is inspired by the idea of work [7], which uses $H^*$-graphs (comprised of vertices with higher degrees$^2$) of an original graph to initially find some cliques, and then expands these cliques to find a portion of maximal cliques from the original graph.

Similarly, we can discover some initial $k$-connected subgraphs using vertices with high degrees. To be specific, we can load into memory the vertices with degrees above a certain level, e.g., $(1 + f) \cdot k$ where $f > 0$, and find $k$-connected subgraphs using these “popular” vertices. The smaller $f$ we choose, the more likely we can discover some $k$-connected subgraphs, but, at the same time, the more time we will spend on finding these initial $k$-connected subgraphs, because more nodes and edges need to be loaded into memory. In the implementation, given a memory pool to hold the vertices and edges, we can choose an $f$ as small as possible on the condition that the memory pool does not overflow if we load all vertices with degree higher than $(1 + f) \cdot k$.

In fact, the heuristic method introduced in this section is reciprocal to the method using materialized views. At the beginning, a system has no materialized views, so some initial $k$-connected subgraphs could be discovered from scratch using the method in this subsection. As the system runs on, more and more materialized views will be available, and the materialized view based method will play a more important role since it is usually more efficient than finding initial $k$-connected subgraphs from scratch.

4.2.3 Expanding existing $k$-connected subgraphs

The third method does not discover $k$-connected subgraphs from scratch. It takes existing $k$-connected subgraphs (possibly produced by the first and second methods) as input, and quickly expand the existing $k$-connected subgraphs in order to find larger ones. The expanding idea is: let a given $k$-connected subgraph be a core, let the core absorb neighbor vertices while keeping itself $k$-connected, stop the absorbing process when the core is not growing fast any more. Here, a neighbor vertex is a vertex not in the core, but is incident on an edge which has the other end in the core. Algorithm 2 illustrates the expanding process. The algorithm steps are self-explanatory. In step 4, the new $G'_s$ is guaranteed to be $k$-connected by Lemma 3. In step 5, $\theta \in [0, 1)$ is a user-defined threshold. The larger $\theta$ is defined, the larger $G'_s$ will be obtained and accordingly the more time the expanding process will take.

**Lemma 3.** Given a simple graph $G$, let $G_s = (V_s, E_s)$ be a $k$-connected subgraph of $G$, let $V_s$ be a set of neighbor vertices of $G_s$ in $G$, then induced subgraph $G[V_s \cup V_a]$ is $k$-connected if and only if $\forall v \in V_a, \deg(v) \geq k$ in $G[V_s \cup V_a]$.

**Proof.** The “only if” part is obvious. We now prove the “if” part. Firstly, according to Theorem 2, we can safely

$^1$There may be a brilliant method to achieve this, but, at the current stage, the problem is open.

$^2$This is a rough idea, interested readers could refer to the paper for a more accurate definition of $H^*$-graph.
Algorithm 2 Expanding a k-connected subgraph

Input: a k-connected subgraph $G_s = (V_s, E_s)$;
Output: a larger k-connected subgraph $G'_s$ ($G'_s$ may stay the same as $G_s$);

1. $G'_s \leftarrow G_s$;
2. repeat
3. let all neighbor vertices of subgraph $G'_s = (V'_s, E'_s)$ be $V_{neighbor}$, generate an induced subgraph $G[V'_s \cup V_{neighbor}]$ from the original graph;
4. repeatedly remove vertices with degree less than $k$ from $G[V'_s \cup V_{neighbor}]$ and assign the result graph as the new $G'_s$ (to be used in the next loop), let the vertices removed in this step be $\Delta V_{neighbor}$;
5. until $\Delta V_{neighbor} > \theta$
6. return $G'_s$;

Figure 2: Expanding the graph to the end

Figure 3: The example of graph reduction

already maximal.

5. EDGE REDUCTION

After contracting k-connected subgraphs into single vertices (in Section 4), the resulting connected components will usually be very dense. In order to efficiently cut the resulting connected components further, it is desirable to use a method not degraded by the large number of edges. In this section, we propose an iterative method based on edge reduction. We first introduce the idea, and then give theoretical foundations and algorithmic solutions for each step. To make the steps clear, we use a running example throughout the whole section.

5.1 Reduction Idea

The reduction idea is to remove vertices by inspecting a sparser subgraph. The reduction includes three steps:

1. Keep the vertex set unchanged and only remove the edges. A graph $G(V, E)$ will be reduced into $G'(V, E')$, where $E' \subseteq E$ and $|E'| \leq i(|V| - 1)$ (here $i \leq k$). We can guarantee that, if any two vertices in $G$ are $k$-connected, then they are $i$-connected in $G'$.  

2. With the above property, we know that all vertices in a $k$-connected subgraph $G_s(V_s, E_s)$ from $G$ are pairwise $i$-connected in $G'$. Therefore, by discovering $i$-connected components from $G'$, we can obtain a vertex super-set $V'_s$ of $V_s$, satisfying any two nodes in $V'_s$ are $i$-connected in $G'$. We can guarantee that $V_s \subseteq V'_s$.

3. Using $V'_s$, we can get a smaller induced subgraph $G'[V'_s]$ from $G$. Thereafter, we may either apply Algorithm 1 on $G'[V'_s]$ directly or repeat the reduction process again on $G'[V'_s]$ using another $i'$ (here $i < i' \leq k$).

Readers are not required to fully understand the three steps at this stage, just need to know the rough picture. We will explain each step in detail in the following sections.

5.2 Theoretical Foundation for Step One
The initiative is to reduce the size of edges in $G$ so that a minimum cut algorithm can be faster. For instance, in the extreme case, $|E|$ is in the size of $|V|^2$. After the reduction, $|E'|$ is in the size of $i|V|$ ($i < k$), at a lower order. The complexity of a typical minimum cut algorithm, due to Stoer and Wagner [27], will be reduced from $O(|V|^3)$ to $O(|V|^2 \log |V|)$, recall the complexity of the algorithm is $O(|E||V| + |V|^2 \log |V|)$. It is obvious that this reduction is especially important when input graph $G$ is dense. Now we come to two questions: (a) Is the edge reduction safe? (b) How to do the reduction?

For the question (a), Lemma 4 can guarantee that: if any two vertices in $G$ are $k$-connected, then they are $i$-connected in $G'$. For the question (b), $G'(V, E')$ can be constructed according to the description of $G_i$ in Lemma 4. The set of spanning forests can be found in $O(|E| + |V|)$ time, due to Nagamochi and Ibaraki [15].

**Lemma 4.** For a graph $G = (V, E)$, simple or multiple, let $F_i = (V, E_i)$ be a spanning forest in $G$ and $F_i$ be a spanning forest in $G - E_i \cup E_{i+1} \cup \cdots \cup E_1$, for $i = 2, 3, \cdots, |E|$, where possibly $E_i = E_{i+1} = \cdots = E_1 = \emptyset$ for some $i$. Then if an induced subgraph $G[V_s]$ ($V_s \subseteq V$) of $G$ is $k$-connected, then, for any $i \leq k$, any two distinct vertices $x, y \in V_s$ is $i$-connected in $G_i = (V, E_i \cup E_{i+1} \cup \cdots \cup E_1)$.

**Proof.** For any two distinct vertices $x, y \in V_s$, considering that the local edge-connectivity between $x, y$ in $G$ is no less than the local edge-connectivity between $x, y$ in $G[V_s]$, given $G[V_s]$ is $k$-connected, we have $\lambda(x, y; G) \geq k$.

Here $\lambda(x, y; G)$ denotes the edge-connectivity between vertices $x, y$ in $G$. According to lemma 2.1 in [16], $\lambda(x, y; G) \geq \min \{\lambda(x, y; G_i) \geq k \geq i\}$, combining with $\lambda(x, y; G) \geq k \geq i$, we have $\lambda(x, y; G) \geq i$. \[\square\]

We use an example to illustrate the process. See Fig. 3, $G_0$ is the original graph, let $k = 5$, so $G[A, B, C, D, E, F]$ is a maximal 5-connected subgraph. After reduction step one, $G_0$ is the reduced graph using $i = 3$. $G_0$ can be denoted as $G_0(V, E_0 \cup E_2 \cup E_3)$. Precisely speaking, $E_0$ is the outer circle except the edge $CD$, $E_2 = \{FA, FB, FC, FD, AE\}$, $E_3 = \{AC, AD, BD, BE\}$. $E_3$ is a spanning tree in $G_0$, $E_2$ is a spanning tree in $G_0 - E_3$, $E_3$ is a spanning tree in $G_0 - E_3 \cup E_2$, and $E_1, E_2, E_3$ are all spanning forests. Since $G_0, G_0 - E_1$ and $G_0 - E_1 \cup E_2$ are all connected, the spanning forests happen to be spanning trees. It is not difficult to verify that any two nodes from $\{A, B, C, D, E, F\}$ are 3-connected in $G_0$.

**5.3 The Problem and Algorithmic Solutions for Step Two**

As to the second step, after $G' = (V, E')$ is obtained from the first step, we want to find all $i$-connected components in $G'$. An $i$-connected component is a set of vertices, and any two vertices in the set are $i$-connected in $G'$. Suppose $G_i = (V_i, E_i)$ is a maximal $k$-connected subgraph in $G$, then all the vertices in $V_i$ fall into a certain $i$-connected component in $G'$, denoted as $V'$. Obviously, $V_i \subseteq V'$. The problem now is to find all these $i$-connected components from $G'$. In other words, $i$-connectivity on $G' = (V, E')$ is an equivalence relation on $V$, and we want to find all non-singleton equivalence classes from $V$ with respect to this equivalence relation.

A straightforward method is to find edge-connectivity for all vertex pairs in $G'$, and then divide the vertices into groups. Lemma 1 guarantees the correctness. Naively, this process needs $\binom{|V|}{2}$ minimum $s-t$ cut computation. Goemory and Hu [9] showed that $n - 1$ minimum $s-t$ can do the job. Their algorithm computes a weighted cut-tree $T$ from $G'$, known as the Goemory-Hu tree, with the property that the edge connectivity between any two vertices $s$ and $t$ in $G'$ exactly equals the weight of the tightest edge in the unique $s-t$ path in $T$. Furthermore, the partition of the vertices produced by removing this edge from $T$ produces a minimum $s-t$ cut to the graph $G'$.

Among the candidate algorithms, one algorithm that is specially suitable to solve the problem is due to Hariharan et al. [11]. Their algorithm uses a graph and a user-specified $k$ as input. The output of the algorithm is a tree $T$ whose nodes represent $k$-connected components. To introduce more, the output is a weighted tree $T$ whose nodes are vertex sets $V_1, V_2, \cdots, V_i$, a partition of $V$, with the property that the connectivity in $G'$ between any two vertices $s \in V_i$ and $t \in V_j$, for $i \neq j$, is equal to the weight of the lightest edge on the path between $V_i$ and $V_j$. Also, two vertices $s$ and $t$ belong to the same $V_i$ for any $i$ if and only if they are at least $k$-connected in $G'$. The complexity of the algorithm is $O(|E| + |V|)$. \[\square\]

Return to the example in Fig. 3, on the reduced graph $G_0$, vertices $A, B, C, D, E, F$ are pairwise 3-connected. They are in the same 3-connected component. Other vertices like $G, H, I$ are singleton 3-connected components, and can be safely pruned. As a result, the only 3-connected equivalent class we can find from $G_0$ is $\{A, B, C, D, E, F\}$.

**5.4 An Example for Step Three**

As to step three, apparently, if $V_s \subseteq V'_s$, then $G[V_s]$ is a subgraph of $G[V'_s]$. We are safe to deal with $G[V'_s]$, a smaller graph compared to $G$, since $G[V'_s]$ have filtered part of the vertices from the original graph $G$. In the example in Fig. 3, the maximal $k$-connected subgraph has vertex set $V_s = \{A, B, C, D, E, F\}$, and the corresponding superset $V'_s$ in $G_0$ is also $\{A, B, C, D, E, F\}$. $G[V'_s]$ and $G[V_s]$ happen to be the same. After we get $G[V'_s]$, we can either run Algorithm 1 to find the real results or repeat the reduction on $G[V'_s]$. In this example, $G[V'_s]$ will not be further reduced, if we repeat the reduction with $i' = 4$ or 5.

**5.5 A Pitfall of Using Graph Reduction**

In the second reduction step, suppose we have got $G_i$ ($i \leq k$) according to Lemma 4, some readers may ask whether we can perform Algorithm 1 on $G_i$ to firstly obtain a set of induced $i$-connected subgraphs, and then find induced subgraphs from $G$ with the vertices in those $i$-connected subgraphs. For example, let $G_i' = (V'_i, E'_i)$ be an induced $i$-connected subgraph in $G_i'$, is it safe to use $G[V'_i]$ as the input for further computation? In other words, does the graph $G_i' = (V'_i, E'_i)$ as a maximal $k$-connected subgraph in $G_i'$, does it mean there must exist an induced $i$-connected subgraph $G_i = (V'_i, E'_i)$ in $G_i'$ satisfying $V_i \subseteq V'_i$? Unfortunately, the answer is no. Refer to Fig. 3, in an induced 3-connected subgraph of $G_i'$ is $\{A, B, D, E, F\}$. Here, vertex $C$ is cut off from the graph, because after vertex $H$ is cut off from the graph, $C$ is no longer 3-connected to vertices $\{A, B, D, E, F\}$. Consequently, in reduction step two, finding $i$-connected components cannot be replaced with finding induced $i$-connected subgraphs.
6. CUT OPTIMIZATION

Given a connected component, there are several cases when we do not need to run a minimum cut algorithm on the connected component. We can tell whether the connected component is $k$-connected or not by inspecting its vertex degrees. This can dramatically improve the algorithm performance. We first list the cases and then explain the rationale afterwards. Let $G_1(V_1, E_1)$ be a connected component, $\Delta(G_1)$, $\delta(G_1)$ be the maximum and minimum vertex degree in $G_1$, $v$ be a vertex in $G_1$.

1. When $G_1$ is simple and $|V_1| \leq k$, i.e. a connected component has no more than $k$ vertices, the component does not have induced $k$-connected subgraphs, and hence can be disregarded.

2. When $\Delta(G_1) < k$, i.e. the maximum degree of the vertices in the connected component $G_1$ is less than $k$, it reflects that the component does not have induced $k$-connected subgraphs.

3. If a vertex $v$ in $G_1$ has $\deg(v) < k$, vertex $v$ can be disregarded from the component. $G_1[V_1 - \{v\}]$ may still have induced $k$-connected subgraphs.

4. If $\delta(G_1) \geq k$ and $\Delta(G_1) \geq \lfloor |V_1|/2 \rfloor$, then the connected component $G_1$ is $k$-connected. We do not need to apply the minimum cut algorithm on $G_1$.

We now explain the rationale behind those optimizations.

- For (1), if component $G_1$ is simple, in any induced subgraph $G_s(V_s, E_s)$ from $G$, separating a node $v$ from $G_s$ requires to remove at most $|V_s| - 1$ edges. (Here, $|V_s| - 1 \leq |V_1| - 1 < k$, because $G_1$ is simple and $|V_1| \leq k$.) In other words, there exists a cut set $E_{cut}$ for $G_s$ with $|E_{cut}| < k$. As a result, $\kappa(G_s) < k$. There is no induced $k$-connected subgraph in $G_1$.

- As to (2), in any induced subgraph $G_s(V_s, E_s)$, separating a node $v$ from $G_s$, requires to remove all $\Delta(G_1)$ edges. Since $\Delta(G_1) < k$, we have $\kappa(G_s) < k$ for a similar reason as (1). Note that (2) also holds for multiple graphs, if $G_1$ is a simple graph, (1) is a special case of (2).

- For (3), the rationale is obvious, removing $v$ is a special light-weighted cut.

- Finally, (4) is supported by Theorem 1 in [5]. We rephrase the theorem as Lemma 5 for easy reference. According to Lemma 5 and $\delta(G_1) \geq k$, we know $G_1$ is $k$-connected if the conditions in (4) are satisfied.

**Lemma 5.** Let $\delta(G)$ be minimum degree among all vertices in $G(V, E)$, if $\delta(G) \geq \lfloor |V|/2 \rfloor$, then $\kappa(G) = \delta(G)$.

All the above four optimizations are designed to avoid performing the minimum cut algorithm in line 3 in Algorithm 1. Condition checks (e.g., checking $|V_1| \leq k$, $\Delta(G_1) < k$) and variable maintenance (e.g., updating $|V_1|$, $\Delta(G_1)$, $\delta(G_1)$, $\deg(v)$) can be done together in $O(|V| + |E|)$ time.

A careful reader may have found that it is not a must to find a minimum cut in line 3 in Algorithm 1. Any cut $E'_{cut}$ with $|E'_{cut}| < k$ can be used to cut $G_1$, and guarantees the correctness of the algorithm. So what is a desirable min-cut algorithm for our problem then? We suggest the minimum cut algorithm due to Stoer and Wagner [27], denoted by the SW algorithm, which provides an early-stop property, and is also reasonably efficient and easy to implement.

We give the SW algorithm in Algorithm 3 and 4 and introduce it briefly. Algorithm 3 is the outer loop, runs $|V| - 1$ times, because after each loop, $|V|$ will be decreased by 1, for the reason that two vertices are merged into one after each loop (see line 5 in Algorithm 4). In each loop from line 2 to line 8 in Algorithm 3, it finds a new min-cut for the current graph (line 3), and compares with the current cut. A smaller cut will be recorded. At the end of this algorithm, the recorded cut will be the minimum cut. The key steps are in Algorithm 4. It first selects a seed vertex, and repeatedly take out other vertices from $|V|$ to join the seed vertex. In each round, the vertex having the highest connectivity with the seed set is selected and removed from $V$.

We give the SW algorithm in Algorithm 3 and 4 and introduce it briefly. Algorithm 3 is the outer loop, runs $|V| - 1$ times, because after each loop, $|V|$ will be decreased by 1, for the reason that two vertices are merged into one after each loop (see line 5 in Algorithm 4). In each loop from line 2 to line 8 in Algorithm 3, it finds a new min-cut for the current graph (line 3), and compares with the current cut. A smaller cut will be recorded. At the end of this algorithm, the recorded cut will be the minimum cut. The key steps are in Algorithm 4. It first selects a seed vertex, and repeatedly take out other vertices from $|V|$ to join the seed vertex. In each round, the vertex having the highest connectivity with the seed set is selected and removed from $V$.

**Algorithm 3 MinimumCut(G)**

**Description:** find a minimum cut for graph $G$

**Input:** a graph $G(V, E)$

**Output:** a min-cut edge set $E_{cut}$

1: initialize $E_{cut} = V$;
2: while $|V| > 1$ do
3: $E_{cut} = $ MinimumCutPhase($G$);
4: if $|E_{cut}| > |E'_{cut}|$ then
5: $E_{cut} = E'_{cut}$;
6: end if
7: end while
8: return $E_{cut}$

**Algorithm 4 MinimumCutPhase(G)**

**Description:** find an $s$-$t$ cut and merge two vertices

**Input:** a graph $G(V, E)$

**Output:** the edge set incident on $V_{last}$

1: randomly choose a vertex $v$, and let $A = \{v\}$;
2: while $A \neq V$ do
3: $E'_{cut}$ add into $A$ the most tightly connected vertex from $V$;
4: end while
5: merging the last two vertices added into $A$;
6: return the edge set $E'_{cut}$ between the last added vertex $v_{last}$ and the rest vertices $V - \{v_{last}\}$;
Finally, we give a combined algorithm in Algorithm 5 to incorporate all speed-up techniques in an overall framework. In Algorithm 5, we restrict ourselves to apply each reduction technique once. The order of the reduction techniques is carefully organized. However, we need to stress that Algorithm 5 is not the only acceptable solution. Each reduction technique may be applied multiple times and the order of some reduction techniques can be exchanged. For example, cut pruning check can be applied every time after a connected component is updated. We can also perform vertex reduction using available k-connected subgraphs first, and then expanding the resulting contracted vertices. Apparently, it is difficult to give an optimal algorithm that best organizes the speed-up techniques, because the effect of speed-up techniques is data-dependent. Nevertheless, Algorithm 5 is still valuable to provide a guideline on how to combine all the speed-up techniques in one framework.

7. EXPERIMENTS

In this section, we report the performance of the basic algorithm and the performance of applying different speed-up techniques on top of the basic algorithm. The results show that the speed-up techniques can improve the performance significantly. All experiments are done on a desktop with Intel(R) Core(TM) 2 Duo CPU E6550 at 2.33GHz and 3GB RAM. The operating system is Windows XP, and code is written in Java.

### 7.1 Datasets

We use three datasets to test the algorithms. Epinions social network (soc-Epinions1) [20], Arxiv GR-QC collaboration network (ca-GrQc) [13], and Gnutella peer-to-peer network (p2p-Gnutella08) [21]. Epinions social network is a who-trust-whom online social network of a general consumer review site (www.Epinions.com). Members of the site decide whether to trust each other. Arxiv GR-QC (General Relativity and Quantum Cosmology) collaboration network is from the e-print arXiv and it covers scientific collaborations between authors according to papers submitted to General Relativity and Quantum Cosmology category. Papers are from January 1993 to April 2003. Gnutella peer-to-peer network data is a snapshot of peer-to-peer file-sharing network in August 8, 2002. All the above datasets are in Stanford Large Network Dataset Collection\(^3\). We give the details of each dataset in terms of number of vertices, edges, and average degrees in Table 1.

### 7.2 Effect of Cut Pruning

In this section, we report the effect of cut pruning (introduced in Section 6) solely without applying vertex reduction and edge reduction. Fig. 4 shows the result. We compare the basic approach (Naive) and the basic approach with cut pruning (NaiPru) on p2p network data and collaboration network data. On both datasets, the pure basic algorithm is rather slow, while after cut pruning, the performance is improved dramatically. When \(k\) becomes larger, the performance of NaiPru is improving as well. The reason is that when \(k\) is larger, more connected components can be pruned. In the following experiments, cut pruning is applied by default so that the baseline approach is not too slow. Cut pruning is orthogonal to vertex reduction and edge reduction.

### 7.3 Effect of Vertex Reduction

In this section, we report the effect of vertex reduction. We test the result on two relatively large datasets collaboration network and Epinions network. Four variant approaches using vertex reduction are tested and compared with the NaiPru approach. Table 2 gives the details of the approaches. Fig. 5 shows the experiment results. On collaboration network data, all four approaches have improved the performance significantly (note that the y-axis is in logarithmic scale, so looks not that impressive). Most of the time, expanding process can further improve the performance, especially when \(k\) is not large, because it is likely to find larger k-connected subgraphs using expansion. When \(k\) is large,\(^3\)

### Table 1: Datasets

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Vertices</th>
<th>Edges</th>
<th>Avg Degree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gnutella P2P network</td>
<td>6301</td>
<td>20777</td>
<td>3.30</td>
</tr>
<tr>
<td>Collaboration network</td>
<td>5242</td>
<td>28980</td>
<td>5.53</td>
</tr>
<tr>
<td>Epinions network</td>
<td>75879</td>
<td>508837</td>
<td>6.71</td>
</tr>
</tbody>
</table>

\(^3\)http://snap.stanford.edu/data/
Table 2: The Meanings of the Approaches

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HeuOly</td>
<td>use only the heuristic method in Section 4.2.2 to find a number of ( k )-connected subgraphs, and then do vertex reduction using these subgraphs;</td>
</tr>
<tr>
<td>HeuExp</td>
<td>use the heuristic method in Section 4.2.2 and the expanding method in Section 4.2.3 to find a number of ( k )-connected subgraphs, and then do vertex reduction</td>
</tr>
<tr>
<td>ViewOly</td>
<td>use only the materialized views (Section 4.2.1) to find a number of ( k )-connected subgraphs, and then do vertex reduction using these views;</td>
</tr>
<tr>
<td>ViewExp</td>
<td>use the materialized views (Section 4.2.1) together with the expanding method in Section 4.2.3</td>
</tr>
</tbody>
</table>

7.4 Effect of Edge Reduction

In this section, we test the effectiveness of edge reduction. As mentioned in Section 5, we can reduce edge iteratively. However, the iteration steps should not be too many, otherwise extra cost will overwhelm the inherent subgraph discovery cost. We compare three approaches with NaiPru, denoted as Edge1, Edge2 and Edge3. Edge1 performs edge reduction once. Edge2 reduces the graph using a \( k' \) \((k' < k)\) firstly, and then \( k \). To generalize the case, we set \( k' = k/2 \). Similarly, Edge3 reduces the graph in three steps, \( k/3, 2k/3 \) and then \( k \). Unlike vertex reduction, we did not test \( k = 6 \) for the collaboration network, neither \( k = 25 \) for the Epinions network. We want to test the case when \( k \) is enough large so that approach Edge3 makes sense. From Fig. 6, we find that, on the network data, Edge1 is usually the best speed-up choice; when \( k = 20 \), Edge2 is slightly better than Edge1. The reason may be that the first step \( k/3 \) can effectively reduce the size of the graph. For all \( k \)'s, Edge3 is the worst choice, even worse than ignoring edge reduction. This confirms that too much edge reduction is even more expensive. On the Epinions data, the result is similar. Edge1 is always better than the other approaches.

7.5 Effect of All Speed-up Techniques

In this section, we report the effect of all the combined speed-up techniques. The BasicOpt approach in this section stands for an approach after applying both vertex reduction and edge reduction on top of the NaiPru method. As to vertex reduction, if there is no materialized views, HeuExp will be used to achieve the largest reduction probability; oth-

![Figure 4: Cut Pruning](image1)

![Figure 6: Edge Reduction](image2)
erwise ViewExp will be used to use materialized views to support vertex reduction. Edge reduction is iterated once in BasicOpt, because most of time, one edge-reduction iteration is the best, though sometimes not as good as twice. On both collaboration network data and Epinions social network data, the BasicOpt approach is up to 10 times faster than the NaiPru approach. Further combining with Fig. 4, BasicOpt is better than the Naive approach in order of magnitudes.

Figure 5: Vertex Reduction

Figure 7: The Combined Effect

8. RELATED WORK
Graph connectivity is a fundamental subject in graph theory. Graph connectivity is closely related to minimum cut, since the minimum cut gives the graph connectivity. A large number of have been done on the design of minimum-cut algorithm [10, 15, 27]. However, these works focus on the global connectivity, i.e. the connectivity of the whole graph, while, in this paper, we aim to find a subgraph with connectivity guarantee. The most similar work to ours is [11], where an algorithm is given to find all pairs of vertices, each of which has a connectivity no less than \( k \), but again, the connectivity is defined on the whole graph, not constrained on a local subgraph. In other works, Yan et al. [29] proposed to find frequent connected subgraphs from a large graph and connectivity is a constraint. In our work, we aim to find maximal connected subgraphs from a given graph. It does not have the \( k \)-edge-connected requirement, and hence algorithms are in a progressive manner and cannot be adapted to our problem. Karypis and Kumar [12] developed a coarsening heuristic for a large graph. The aim is to reduce the input graph scale, similar to our graph reduction, but the techniques are different. Finally, in presence of many deterministic min-cut algorithms, Chekuri et al. [6] showed the algorithms in [10] and [15] are fast in practice, and may be good candidates to resort to.

On the other hand, works on extracting subgraphs from a given graph can be divided into two categories: explicit and implicit. In explicit works, such as, quasi-clique [30, 1], k-core [24], k-plex [23], a structure with certain property is predefined, and then the rest work is to design efficient algorithms to discover all the subgraphs with the structure requirement. In implicit works, some propose objective functions first, such as modularity [17], normalized cut [25], and then partition the graph into a number of parts, here a good partition usually maximizes or minimizes the objective functions; Some works define neighbourhood distance, such as propinquity [31], structure closeness [28], and then group nearby nodes within a distance threshold around a given node to form a group; Some borrow the idea of Markov Clustering [22, 19] to repeat random walk for a few rounds until self-organized clusters turn up. Different from the implicit models, the maximal \( k \)-connected subgraphs we aim to find is explicitly defined.

9. CONCLUSIONS AND FUTURE WORK
In this paper, we have discussed how to find maximal \( k \)-edge-connected subgraphs from large graphs. We have proposed a basic cut-based approach and develop several important speed-up techniques: vertex reduction, edge reduction and optimizing the cut algorithm. We have conducted extensive experiments to test the performance of the speed-up
techniques. Compared with the basic approach, experimental results show that using speed-up methods can dramatically improve the performance. Vertex reduction is very effective when there are suitable materialized views, because the size of the original graph can be reduced significantly. The effect of edge reduction is moderate, not remarkable, because after discovering the $k$-connected components, we still need to find maximal $k$-connected subgraphs from those $k$-connected components. The cut pruning is very effective, and it is also easy to implement. One direction for future work is to design external memory algorithms to find maximal $k$-connected subgraphs from massive graphs, because some real graphs are too large to fit into memory.

Acknowledgments

This work is supported by the grant of Australian Research Council Discovery Project No. DP120102627 and the grant of Research Grants Council of the Hong Kong SAR, China, No. 419109. We would like to thank anonymous reviewers for their helpful comments.

10. REFERENCES


