Structural Community Detection in Big Graphs

by

DONG WEN

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Centre for Artificial Intelligence (CAI)
Faculty of Engineering and Information Technology (FEIT)
University of Technology Sydney (UTS)

June, 2018

CERTIFICATE OF AUTHORSHIP/ORIGINALITY

I certify that the work in this thesis has not previously been submitted for a degree nor has it been submitted as part of requirements for a degree except as fully acknowledged within the text.

I also certify that the thesis has been written by me. Any help that I have received in my research work and the preparation of the thesis itself has been acknowledged. In addition, I certify that all information sources and literature used are indicated in the thesis.

Production Note:
Signature removed prior to publication.

Signature of Candidate

ACKNOWLEDGEMENTS

First, I would like to deliver my sincere gratitude to my supervisor Dr. Lu Qin for his continuous support and guidance of my PhD career. He is professional, efficient, patient and diligent. His valuable ideas always guided me and extended my knowledge not only in the database area, but also about the practical research skills. Additionally, Lu is a good mentor and friend for me. He is selfless and glad to share his own experience with me. Thanks to his encouragement, I am always positive and brave when experiencing challenges and even failures. This thesis could not reach its present form without his illuminating instructions.

Secondly, I would like to express my great gratitude to my co-supervisor Prof. Ying Zhang for his guidance and advice, especially for his strong confidence in me and support for my academic career. He gave me many wonderful ideas and inspirations. His guidance significantly extended my knowledge and helped me pursue a correct research direction all the time. He is also visionary, and often encouraged and guided me to optimize the plan for the career development.

Thirdly, I would like to thank Prof. Xuemin Lin and Dr. Lijun Chang for supporting the works in this thesis, as most of the works were conducted in collaboration with them. I thank Prof. Lin for offering an interesting but rigorous

research environment. I learned the characteristics of an excellent researcher from Prof. Lin — passion, preciseness and earnest. I thank Dr. Chang for his brilliant ideas and confidence in me. His wonderful research works always inspired me, and he gave me many accurate and valuable suggestions to improve the quality of the works in this thesis.

I would also like to thank the following people: Prof. Jeffrey Xu Yu, Dr. Wenjie Zhang, Dr. Xin Cao, Dr. Zengfeng Huang, Dr. Ling Chen, Dr. Xiaoyang Wang, Dr. Shiyu Yang, for sharing valuable ideas and experiences. Thanks to Dr. Fan Zhang, Dr. Long Yuan, Dr. Longbin Lai, Dr. Xing Feng, Dr. Xiang Wang, Dr. Jianye Yang, Dr. Fei Bi, Mr. Xubo Wang, Mr. Wei Li, Mr. Haida Zhang, Ms. Chen Zhang, Dr. Qing Bing, Dr. Shenlu Wang, Mr. Yang Yang, Mr. Kai Wang, Mr. You Peng, Mr. Hanchen Wang, Mr. Xuefeng Chen, Mr. Boge Liu, Mr. Jiahui Yang, Dr. Wei Wu, Dr. Qian Zhang, Dr. Zhaofeng Su, Mr. Dian Ouyang, Mr. Wentao Li, Mr. Bohua Yang, Mr. Mingjie Li, Ms. Conggai Li, Mr. Wei Song, for sharing the happiness with me in my PhD career.

Last but not least, I would like to thank my father Mr. Rongxiao Wen and my mother Mrs. Tiefang Liu, for bringing me a wonderful life, and other relatives for their understanding, encouragement and love. Thanks to my wife Mrs. Qing Lin, for her company and support in my PhD study.

ABSTRACT

Community detection in graphs is a fundamental problem widely experienced across industries. Given a graph structure, one popular method to identify communities is classifying the vertices, which is formally named graph clustering. Additionally, community structures are always dense and highly connected in graphs. There are also a large number of research works focusing on mining cohesive subgraphs for community detection. Even though the community detection problem is extensively studied, challenges still exist. With the development of social media, graphs are highly dynamic, and the size of graphs is sharply increasing. The large time and space cost of traditional solutions may hardly be endured in big and dynamic graphs.

In this thesis, we propose an index-based algorithm for the structural graph clustering (SCAN). Based on the proposed index structure, the time expended to compute structural clustering depends only on the result size, not on the size of the original graph. The space complexity of the index is bounded by $O(m)$, where m is the number of edges in the graph. We also propose algorithms and several optimization techniques for maintaining our index in dynamic graphs.

For the cohesive subgraph detection, we study both degree-constrained (k-core) and connectivity-constrained (k-VCC) cohesive subgraph metrics. A k-core is a maximal connected subgraph in which each vertex has degree at least k. We study I/O efficient core decomposition following a semi-external model, which only allows vertex information to be loaded in memory. We propose an optimized

I/O efficient algorithm for both core decomposition and core maintenance. In addition, we extend our algorithm to compute the graph degeneracy order, which is an important graph problem that is highly related to core decomposition.

A k-vertex connected component (k-VCC) is a connected subgraph in which the removal of any $k-1$ vertices will not disconnect the subgraph. A k-VCC has many outstanding structural properties, such as high cohesiveness, high robustness, and subgraph overlapping. The state-of-the-art solution enumerates all k-VCCs following a partition-based framework. It requires high computational cost in connectivity tests. We prove the upper bound of the number of partitions, which implies the polynomial running time of this framework. We propose two effective optimization strategies, namely neighbor sweep and group sweep, to largely reduce the number of local connectivity tests.

We conducted extensive performance studies using several large real-world datasets to show the efficiency and effectiveness of all our approaches.

PUBLICATIONS

- ***Dong Wen***, *Lu Qin, Xuemin Lin, Ying Zhang, and Lijun Chang, Enumerating k-Vertex Connected Components in Large Graphs, in submission.*
- ***Dong Wen***, *Lu Qin, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu, I/O Efficient Core Graph Decomposition: Application to Degeneracy Ordering, TKDE, revision submitted.*
- ***Dong Wen***, *Lu Qin, Ying Zhang, Lijun Chang, and Xuemin Lin, Efficient Structural Graph Clustering: an Index-Based Approach, to appear in PVLDB 2018.*
- *Lingxi Yue,* ***Dong Wen***, *Lizhen Cui, Lu Qin, and Yongqing Zheng, K-Connected Cores Computation in Large Dual Networks, to appear in DASFAA 2018.*
- ***Dong Wen***, *Lu Qin, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu, I/O Efficient Core Graph Decomposition at Web Scale, ICDE2016, Best Paper Award.*

TABLE OF CONTENT

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION

Graphs have been widely used to represent the relationships of entities in the real world due to its strong expressive power. Given a graph $G = (V, E)$, vertices in V represent the entities of interest and edges in E represent the relationships between the entities. With the proliferation of graph applications, research efforts have been devoted to many fundamental problems in mining and analyzing graph data; those efforts identify community detection as a fundamental problem. Generally, the community detection aims to compute all communities in a given graph. In the literature, the methods for community detection can be roughly categorized into two classes. The first one is graph clustering [12, 40, 62, 67]. It globally classifies the vertices in the given graph based on a pre-defined function or other structural rules. The other kind of methods is cohesive subgraph detection, given that the community structures in the graph are always dense and highly-connected [82]. These methods directly compute a set of cohesive subgraphs based on a pre-defined cohesive metric.

Even though the community detection problem has been studied extensively, several challenges still exist. An urgent one is the large and constantly growing graph size. For example, the Facebook social network contains 1.4 billion daily

active users on average for December 2017 [1]; and a sub-domain of the web graph Clueweb contains 978.5 million vertices and 42.6 billion edges [2]. A traditional algorithm may incur high computational cost and space cost in such graphs. Additionally, most of real-world graphs are highly dynamic. For example, there are about 500 million tweets on average per day in Twitter [3]. The algorithm should be designed to handle dynamic graphs.

In this thesis, we study the community detection problem given only the graph structure. We propose an index-based algorithm for structural graph clustering (SCAN) based community detection, and for the cohesive subgraph based community detection, we study both degree-constrained (k-core) and connectivity-constrained (k-VCC) cohesive metrics.

Note that there are also several other cohesive subgraph models, such as clique [20], k-truss [23] and k-edge connected component [17]. More details can be founded in Chapter 2. In the literature, each of these models has distinctive structural property and corresponding algorithmic efficiency. In this thesis, we select three fundamental models, and only focus on the improvement of algorithmic efficiency and scalability.

1.1 Structural Graph Clustering

A graph cluster is a group of vertices that are densely connected within a group and sparsely connected to vertices outside that group. Graph clustering mainly aims to detect all clusters in a given graph. In addition to community detection, such analysis is required for metabolic networks [37], creating relevant search results for web crawlers, and identifying research groups across collaborating

[1] https://newsroom.fb.com/company-info/
[2] http://law.di.unimi.it/datasets.php
[3] http://www.internetlivestats.com/twitter-statistics/

networks [99].

While detecting all clusters is important, also worthwhile is identifying the specific role—either hub or outlier—of each vertex that is not a member of any cluster. Hubs are vertices that bridge different clusters, and outliers are vertices that do not. Distinguishing between hubs and outliers is important for mining various complex networks [90, 43]. Normally, hubs are regarded as influential vertices, and outliers are treated as noise.

Structural Graph Clustering. Many different graph clustering methods are proposed in the literature. They include the modularity-based methods [63, 73], graph partitioning [72, 25, 81] and the density-based methods [42]. However, most of these clustering methods only focus on computing clusters, ignoring identification of hubs and outliers. To handle this issue, a *structural graph clustering* (SCAN) method is proposed in [90]. Its basic premise is that two vertices belong to the same cluster if they are similar enough.

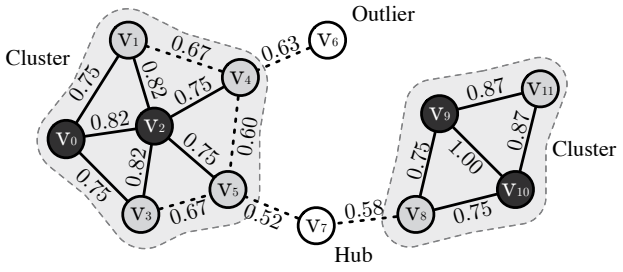


Figure 1.1: Clusters, hubs and outliers under $\epsilon = 0.7, \mu = 4$

SCAN defines the *structural similarity* between adjacent vertices, with two vertices being considered similar if their structural similarity is not less than a given parameter ϵ. To construct clusters, SCAN detects a special kind of vertex, named *core*. A core is a vertex that is neighbored closely by many similar vertices and is regarded as the seed of a cluster. The number of similar neighbors

for a core is evaluated by a given parameter μ. A cluster is constructed by a core expanding to all vertices that are structurally similar to that core. SCAN successfully finds all these clusters, and also the hubs and outliers. If a vertex does not belong to any cluster, it is a hub if its neighbors belong to more than one cluster, and an outlier otherwise. Fig. 1.1 gives an example of graph clustering. In it, two clusters are colored gray, and cores are colored black. Hubs and outliers are also labeled.

Existing Solutions. The effectiveness of structural graph clustering across many applications means that plenty of related researches have been proposed. The original algorithm for SCAN is proposed in [90]; for the ease of presentation, we also use the generic term SCAN to represent this algorithm. It iteratively processes each vertex u that has not been assigned to any cluster. If u is a core, it creates a cluster containing u and recursively adds the similar vertices for cores into the cluster. However, the algorithm needs to compute the structural similarity for every pair of adjacent vertices; this requires high computational cost and does not scale to large graphs. Several methods are proposed for overcoming this drawback. In one method, SCAN++ [74] defines the vertex set containing only vertices that are two hops away from a given vertex. This helps SCAN++ compute fewer structural similarities than SCAN. LinkSCAN* [50] improves the efficiency of SCAN via sampling edges and obtains an approximate result of SCAN.

The state-of-the-art solution for improving SCAN's algorithm efficiency is pSCAN [15]. Under this method, identification of all cores is the key to structural graph clustering. To reduce the number of similarity computations, pSCAN maintains an upper bound (*effective-degree*) and a lower bound (*similar-degree*) for the number of similar neighbors of each vertex. pSCAN always processes the vertex that has the largest effective-degree, which means that vertex has

a high probability of being a core. pSCAN avoids a large number of similarity computations and is significantly faster than previous methods.

Several other methods address the problem of structural graph clustering across different computational environments. A parallel version of SCAN is studied in [97]; the algorithm for SCAN on multicore CPUs is studied in [54]. More details about related work are summarized in Chapter 2.

Motivation. Even though these various SCAN methods can successfully compute clusters, hubs, and outliers, several challenges remain:

- *Parameters Tuning.* The results heavily depend on two sensitive input parameters, μ and ϵ, and the optimal parameter setting is dependent on different graph properties and user requirements. To obtain reasonable clusters, users may need to run an algorithm several times to tune the parameters. Therefore, efficiently computing the clusters given to each parameter setting is a critical factor.

- *Query Efficiency.* pSCAN proposes several pruning rules to improve efficiency. However, it still needs to compute the structural similarity for every pair of adjacent vertices in the worst-case scenario. At a minimum, it needs to process all of a graph's vertices and their adjacent edges to obtain the clusters, even if there exist only a small number of vertices belonging to clusters.

- *Network Update.* Many real-world networks are frequently updated. The clustering results may change when an edge is inserted or removed. Solutions for structural graph clustering should consider handling dynamic graphs.

Our Solution. Motivated by the above challenges, we propose a novel index structure, named GS*-Index, for structural graph clustering. GS*-Index has two

main parts: *core-orders* and *neighbor-orders*. Given parameters ϵ and μ, we can easily obtain all cores with the help of core-orders, while neighbor-orders help us group all cores and add non-core vertices into each cluster. The space complexity of GS*-Index is bounded by $O(m)$ and the time complexity to construct GS*-Index is bounded by $O((m + \alpha) \cdot \log n)$, where n is the number of vertices, m is the number of edges and α is the graph arboricity [22].

Based on GS*-Index, we propose an efficient algorithm to answer the query for any possible ϵ and μ. We compute all clusters in $O(\sum_{C \in \mathbb{C}} |E_C|)$ time complexity, where $\mathbb{C}$ is the result set of all clusters and $|E_C|$ is the number of edges in a specific cluster C in $\mathbb{C}$. In other words, the running time of our algorithm is only dependent on the result size, and not on the size of the original graph.

Most real-world networks frequently update. Therefore, we provide algorithms for updating GS*-Index when an edge is inserted or removed. We further propose techniques for improving the algorithmic efficiency.

Contributions. The main contributions in response to the SCAN challenges are listed as follows:

- *The first index-based algorithm for structural graph clustering.* We propose an effective and flexible index structure, named GS*-Index, for structural graph clustering. To the best of our knowledge, this is the first index-based solution for the structural graph clustering problem. The size of GS*-Index can be well bounded by $O(m)$.

- *Efficient query processing.* Based on our GS*-Index, we propose an efficient algorithm for answering the query for any possible ϵ and μ. The time complexity is linear to the number of edges in the resulting clusters.

- *Optimized algorithms for index maintenance.* We propose algorithms for maintaining our proposed index when graphs update. Several optimiza-

tions are proposed for achieving significant algorithmic speedup.

- *Extensive performance studies in real-world networks.* We do extensive experiments for all our proposed algorithms in 10 real-world datasets, one of which contains more than 1 billion edges. The results demonstrate that our final algorithm can achieve several orders of magnitude speedup in query processing compared to the state-of-the-art algorithm.

The details of this work are presented in Chapter 3.

1.2 Degree-Based Cohesive Subgraph Detection

Given a graph G, a k-core of G is a maximal subgraph of G such that all the vertices in the subgraph have a degree of at least k [69]. The problem of computing the k-core of a graph has been recently studied [19, 68, 47, 24]. For each vertex v in G, the core number of v denotes the largest k such that v is contained in a k-core. The core decomposition problem computes the core numbers for all vertices in G. Given the core decomposition of a graph G, the k-core of G for all possible k values can be easily obtained. There is a linear time in-memory algorithm, devised by Batagelj and Zaversnik [9], to compute core numbers of all vertices.

Applications. Core decomposition is widely adopted in many real-world applications, such as community detection [36, 24, 79], network topology analysis [69, 5], network visualization [4, 3], protein-protein network analysis [2, 7], and system structure analysis [93]. In addition, many researches are devoted to the core decomposition for specific kinds of networks [26, 53, 65, 59, 41, 11]. Moreover, due to the elegant structural property of a k-core and the linear solution for core decomposition, a large number of graph problems use core decomposition as a subroutine or a preprocessing step, such as clique finding [8], dense subgraph

discovery [6, 66], approximation of betweeness scores [38], and some variants of community search problems [76, 46].

Motivation. Despite the large amount of applications for core decomposition in various networks, most of the solutions for core decomposition assume that the graph is resident in the main memory of a machine. Nevertheless, many real-world graphs are big and may not reside entirely in the main memory. In the literature, the only solution to study I/O efficient core decomposition is EMCore proposed in [19], which allows the graph to be partially loaded in the main memory. EMCore adopts a partition-based approach and partitions are loaded into main memory whenever necessary. However, EMCore cannot bound the size of the memory usage and to process many real-world graphs, EMCore still loads most edges of the graph in the memory. This makes EMCore unscalable to handle web-scale graphs. In addition, many real-world graphs are usually dynamically updating. The complex structure used in EMCore makes it difficult to handle graph updates incrementally.

Our Solution. In this thesis, we address the drawbacks of the existing solutions for core decomposition and propose new algorithms with a guaranteed memory bound. Specifically, we adopt a semi-external model. It assumes that the vertices of the graph, each of which is associated with a small constant amount of information, can be loaded in main memory while the edges are stored on disk. We find that this assumption is practical in a large number of real-world web-scale graphs, and widely adopted to handle other graph problems [94, 95, 51]. Based on such an assumption, we are able to handle core decomposition I/O efficiently using very simple structures and data access mechanisms. The I/O complexity of our algorithm is $O(\frac{l \cdot (m+n)}{B})$ where l is the iteration number, m and n are the numbers of vertices and edges of the graph and B is the disk block size. We refer to our proposed approace as "I/O efficient" because (1) l is quite small in

practice and (2) in most of the iterations, we only need to access a small part of the graph on disk. For example, to process a dataset with 978.5 million vertices and 42.6 billion edges used in our experiments, the total number of I/Os used by our algorithm is only 3 times the value of $\frac{m+n}{B}$. The semi-external model can also be used to handle graph updates.

Degeneracy Ordering. We also extend our I/O efficient core decomposition algorithm for degeneracy ordering, which is highly related to core decomposition. It is a basic graph problem in graph theory [32]. The degeneracy order of a graph G is a total order of vertices in the graph such that every vertex has limited number of neighbors (denoted as $d(G)$) in its right-side vertices in the order. Degeneracy ordering is applied in many applications such as graph coloring [56] and graph clustering [34, 18]. In graph coloring, if we follow the degeneracy order to color vertices in the graph, we can guarantee that the number of colors used is bounded by $d(G) + 1$ [56]. In graph clustering, a typical problem is to find a subgraph with maximum density. If we construct subgraphs following the degeneracy order, we can find a 2-approximation solution to this problem [18]. Some other applications can be found in [32].

In the literature, all algorithms for degeneracy ordering assume that the graph can reside in memory. We find that, under the semi-external model, the degeneracy order can be computed based on our I/O efficient algorithm for core decomposition, and we can also maintain the degeneracy order incrementally on dynamic graphs based on our I/O efficient core maintenance algorithm.

Contributions. In the following, we summarize the main contributions of this work.

(1) The first I/O efficient core decomposition algorithm with a memory guarantee. We propose an I/O efficient core decomposition algorithm following the semi-external model. Our algorithm only keeps the core numbers of vertices in

memory and updates the core numbers iteratively until convergence. In each iteration, we only require sequential scans of edges on disk. To the best of our knowledge, this is the first work for I/O efficient core decomposition with a memory guarantee.

(2) Several optimization strategies to largely reduce the I/O and CPU cost. Through further analysis, we observe that when the number of iterations increases, only a very small proportion of vertices have their core numbers updated in each iteration, and thus scanning all edges on disk in each iteration will result in a large number of waste I/O and CPU time. Therefore, we propose optimization strategies to reduce this cost. Our first strategy is based on the observation that the update of the core number of a vertex should be triggered by the update of the core number of at least one of its neighbors in the graph. Our second strategy further maintains more vertex information. As a result, we can completely avoid waste I/Os and core number computations, in the sense that each I/O is used in a core number computation that is guaranteed to update the core number of the corresponding vertex.

(3) The first I/O efficient core decomposition algorithm to handle graph updates. We consider dynamic graphs with edge deletion and insertion. Our semi-external algorithm can naturally support edge deletion with a simple algorithm modification. For edge insertion, we first take advantage of some graph properties already used in existing in-memory algorithms [68, 47] to handle graph updates for core decomposition. We propose a two-phase semi-external algorithm to handle edge insertion using these graph properties. We further explore some new graph properties, and propose a new one-phase semi-external algorithm to largely reduce the I/O and CPU time for edge insertion. To the best of our knowledge, this is the first work for I/O efficient core maintenance on dynamic graphs.

(4) The first I/O efficient algorithms for degeneracy order computation and

maintenance. We propose I/O efficient algorithms for degeneracy order computation and degeneracy order maintenance respectively under the semi-external model. We significantly improve their efficiency by making use of our core decomposition algorithm and core maintenance algorithm respectively. To the best of our knowledge, this is the first work for I/O efficient degeneracy order computation and maintenance.

(5) Extensive performance studies. We conduct extensive performance studies using 12 real graphs with various graph properties to demonstrate the efficiency of our algorithms. We compare our algorithm, for memory-resident graphs, with EMCore [19] and the in-memory algorithm [9]. Both our core decomposition and core maintenance algorithms are much faster and use much less memory than EMCore. Our algorithms for degeneracy order computation and maintenance are also efficient and scalable to handle big graphs with bounded memory under the semi-external model.

The details of this work are presented in Chapter 4.

1.3 Connectivity-Based Cohesive Subgraph Detection

A k-vertex connected component (k-VCC), also named *k-component* [78] or *k-block* [55], is *a maximal connected subgraph in which the removal of any $k-1$ vertices cannot disconnect the subgraph.* Given a graph G and an integer k, the problem of enumerating k-VCCs is to find all k-VCCs in G [48]. Fig. 1.2 shows an example; there are four 4-VCCs, namely G_1, G_2, G_3, and G_4 in G. The subgraph formed by the union of G_1 and G_2 is not a k-VCC, because it will be disconnected by removing two vertices a and b.

Vertex connectivity has been proved as an powerful and robust metric to

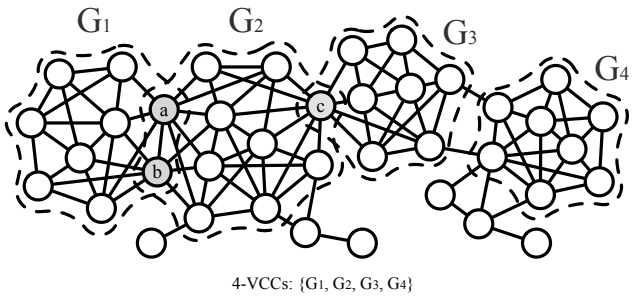


Figure 1.2: Cohesive subgraphs in graph G.

evaluate the cohesiveness of a social group [61, 87]. A k-VCC has following outstanding structural properties:

- *High Cohesiveness.* Given any graph G, Whitney Theorem [88] proves the vertex connectivity of G is not larger than the edge connectivity and the minimum degree. This implies a k-VCC is nested in a k-edge connected component(k-ECC) [98] and a k-core [9]. Therefore, a k-VCC is generally more cohesive and inherits all the structural properties of a k-ECC and a k-core.

- *High Robustness.* Menger's Theorem [57] implies that any pair of vertices in a k-VCC is joined by at least k vertex-independent paths. Each k-VCC cannot be disconnected by removing any $k-1$ vertices.

- *Bounded Diameter.* Small diameter is considered as an important feature for a good community in [27]. The diameter of a k-VCC $G'(V', E')$ is bounded by $\lceil \frac{|V'|-1}{\kappa(G')} \rceil$ where $\kappa(G')$ is the vertex connectivity of G' [14].

- *Subgraph Overlapping.* Community overlap is regarded as an important feature of many real-world complex networks [89]. The vertex overlap is allowed in k-VCC detection, and the number of overlapped vertices between two k-VCCs is bounded by the parameter k.

- *Bounded Subgraph Number.* Even with overlapping, the number of k-VCCs is bounded by $n/2$, where n is the number of vertices in the graph. The detailed proof can be found in Section 5.4. This indicates that redundancies in the k-VCCs are limited.

Enumerating all k-VCCs has many applications. For example, in social network, computing all k-VCCs can identify communities of highly related users, and provide valuable information for recommendation systems and advertisement platforms. In a co-authorship network, a cohesive subgraph may be a research group. Some researchers may participate in several groups and perform as overlapped entities. This problem can also be used to visualize the graph structure [78]. Motivated by the elegant properties and a wide range of applications, the problem of k-VCC enumeration has been studied recently [48, 78, 75].

Existing Solutions and Drawbacks. The state-of-the-art solution for computing all k-VCCs in a given graph G is proposed in [48]. This algorithm computes a vertex cut with fewer than k vertices in G. Here, a vertex cut of G is a set of vertices, the removal of which disconnects the graph. Based on the vertex cut, G is partitioned into overlapped subgraphs, each of which contains all the vertices in the cut along with their incident edges. The algorithm recursively partitions each of the subgraphs until no such cut exists. In this way, they compute all k-VCCs. For example, suppose the graph G is the union of G_1 and G_2 in Fig. 1.2. Given $k = 4$, we can find a vertex cut containing two vertices a and b. Thus we partition the graph into two subgraphs G_1 and G_2 that overlap two vertices a, b and an edge (a, b). Since neither G_1 nor G_2 has any vertex cut with fewer then k vertices, we obtain G_1 and G_2 as the final k-VCCs.

Even though [48] successfully obtains all k-VCCs in a graph G, several challenges still remain. When performing a partition operation, overlapped verices are duplicated, and the total number of partitions can be very large. Addi-

tionally, the most crucial operation in the algorithm is called local connectivity testing, which given two vertices u and v, tests whether u and v can be disconnected in two components by removing at most $k-1$ vertices from G. To find a vertex cut with fewer than k vertices, we need to conduct local connectivity testing between a source vertex s and each of other vertices v in G in the worst case; this requires high computational cost and makes it not scalable to large graphs.

Our Approaches and Contributions. In this thesis, we adopt the same idea as [48] to implement a cut-based solution of k-VCC enumeration. Given a graph G and an integer k, we prove that both the number of overlapped partitions and the number of k-VCCs are not larger than $n/2$, which indicates the polynomial running time of the algorithm. We observe *the key to improving algorithmic efficiency is to reduce the number of local connectivity testings in a graph.* Given a source vertex s, if we can avoid testing the local connectivity between s and a certain vertex v, we call it as we can *sweep vertex* v. We propose two strategies to sweep vertices.

- *Neighbor Sweep.* If a vertex has certain properties, all its neighbors can be swept. Therefore, we call this strategy neighbor sweep. Moreover, we maintain a deposit value for each vertex, and once we finish testing or sweep a vertex, we increase the deposit values for its neighbors. If the deposit value of a vertex satisfies certain condition, such vertex can also be swept.

- *Group Sweep.* We introduce a method to divide vertices in a graph into disjoint groups. If a vertex in a group has certain properties, vertices in the whole group can be swept. We call this strategy group sweep. Moreover, we maintain a group deposit value for each group. Once we test or sweep a vertex in the group, we increase the corresponding group deposit value. If

the group deposit value satisfies certain conditions, vertices in such whole group can also be swept.

Even though these two strategies are studied independently, they can be used together and boost the effectiveness of each other. With these two vertex sweep strategies, we can significantly reduce the number of local connectivity testings in the algorithm. Experimental results show the excellent performance of our sweep strategies. More details can be found in Chapter 5.

Contributions. We make the following contributions.

- *A polynomial time algorithm based on overlapped graph partition.* Given a graph G and an integer k, we prove that the sum of the number of overlapped partitions and the number of k-VCCs is less than number of vertices in G, which indicates a polynomial running time for the algorithm proposed in [48].
- *Two effective pruning strategies.* We design two pruning strategies, namely neighbor sweep and group sweep, to largely reduce the number of local connectivity testings and thus significantly speed up the algorithm.
- *Extensive performance studies.* We conduct extensive performance studies on 10 real large graphs to demonstrate the efficiency of our proposed algorithms.

The details of this work are presented in Chapter 5.

1.4 Graph Model

In this thesis, we consider an undirected and unweighted graph $G(V, E)$, where $V(G)$ represents the set of vertices and $E(G)$ represents the set of edges in G.

We denote the number of vertices $|V|$ and the number of edges $|E|$ by n and m respectively. For each vertex u, the open neighborhood of u, denoted by $N(u, G)$, is the set of neighbors of u, i.e., $N(u, G) = \{v \in V(G) | (u, v) \in E(G)\}$. When the context is clear, we use the term neighbor for short. The degree of a vertex $u \in V(G)$, denoted by $deg(u, G)$, is the number of neighbors of u in G, i.e., $deg(u, G) = |N(u, G)|$. For simplicity, we use $N(u)$ and $deg(u)$ to denote $N(u, G)$ and $deg(u, G)$ respectively. A graph G' is a subgraph of G, denoted by $G' \subseteq G$, if $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$. Given a set of vertices $V_c \subseteq V$, the induced subgraph of V_c, denoted by $G(V_c)$, is a subgraph of G such that $G(V_c) = (V_c, \{(u, v) \in E(G) | u, v \in V_c\})$. For any two subgraphs G' and G'' of G, we use $G' \cup G''$ to denote the union of G' and G'', i.e., $G' \cup G'' = (V(G') \cup V(G'), E(G') \cup E(G''))$. Several frequently used notations are summarized in Table 1.1.

Symbol	Description
$N(u)$	the open neighbors of vertex u
$N[u]$	the structural neighbors of vertex u
$deg(u)$	the number of open neighbors of vertex u
$deg[u]$	the number of structural neighbors of vertex u
$G(V_c)$	the induced subgraph of vertex set V_c
$N_\epsilon[u]$	ϵ-neighbors of vertex u
d_{max}	the maximum degree of graph G
$\vec{deg}(u)$	the backward degree of vertex u in a vertex order
$core(u)$	the core number of vertex u
k_{max}	the maximum core number of graph G
$d(G)$	the degeneracy of graph G
$\kappa(G)$	the vertex connectivity of graph G
$\kappa'(G)$	the edge connectivity of graph G
$\delta(G)$	the minimum degree of graph G
k-VCC	k-vertex connected component
k-ECC	k-edge connected component

Table 1.1: Notations

Based on the aforementioned graph model, we formally summary all research problems in this thesis.

- Given a graph G, an integer parameter μ and a probability threshold ϵ, we aim to compute all clusters in G based on the SCAN model.
- Given a graph G, we aim to compute the core number for each vertex.
- Given a graph G and an integer k, we aim to compute all k-VCCs in G.

Chapter 2

LITERATURE REVIEW

2.1 Graph Clustering

Structural Graph Clustering. The original algorithm for SCAN is proposed in [90]. It successfully identifies not only clusters but also hubs and outliers. However, SCAN computes the structural similarities for all pairs of adjacent vertices, which requires high computational cost and is not scalable to large graphs. To alleviate this problem, [74] proposes an algorithm named SCAN++. It is based on an intuitive idea that it is highly probable that many common neighbors exist between a vertex and its two-hop-away vertices. [15] proposes the algorithm pSCAN, which prior processes vertex with large probabilities as cores. An approximate solution, LinkSCAN* is proposed in [50]. It improves algorithm efficiency by sampling edges.

Other researchers are working to parallelize SCAN. [97] provides a parallel structural graph clustering algorithm based on MapReduce. [54] proposes an algorithm, named anySCAN, which progressively produces an approximate result for clustering on multicore CPUs.

There also exist some parameter-free methods for SCAN, mainly focusing on

computing a suitable ϵ during the algorithm. [13] proposes the algorithms SCOT and HintClus. SCOT is used to compute a sequence containing information for all ϵ-clusters, while HintClus computes hierarchical clusters. gSkeletonClu [77] uses a weighted version of structural similarity and computes clusters based on a tree-decomposition-based method. SHRINK [39] computes clusters by combining structural similarity and modularity-based methods.

Other Graph Clustering Methods. Other graph clustering methods have also been studied in the literature; they include graph partitioning [72, 25, 81], the modularity-based methods [63, 73], and the density-based method [42]. More details can be found in a survey [45, 82].

2.2 Cohesive Subgraph Detection

Efficiently computing cohesive subgraphs, based on a designated metric, has drawn a large number of attentions recently. [16, 20] propose algorithms for maximal clique problem. However, the definition of clique is too strict. For relaxation, some clique-like metrics are proposed. These metrics can be roughly classified into three categories, 1) global cohesiveness, 2) local degree and triangulation, and 3) connectivity cohesiveness.

2.2.1 Global Cohesiveness

[52] defines an s-clique model to relax the clique model by allowing the distance between two vertices to be at most s, i.e., there are at most $s-1$ intermediate vertices in the shortest path. However, it does not require all intermediate vertices are in the s-clique itself. [58] proposes an s-club model requiring that all intermediate vertices are in the same s-club. In addition, k-plex allows each vertex in such subgraph can miss at most k neighbors [10, 70]. Quasi-clique

is a subgraph with n vertices and at least $\gamma * \binom{n}{2}$ edges [92]. These kinds of metrics globally require the graph to satisfy a designated density or other certain criterions.

2.2.2 Local Degree and Triangulation

Degree-Based Metrics. k-core is first introduced as a tool for complex network analysis in [69]. Batagelj and Zaversnik [9] give an linear in-memory algorithm for core decomposition, which is presented in Section 4.3. [19] proposes an I/O efficient algorithm for core decomposition. [60] gives a distributed algorithm for core decomposition. [68] and [47] propose in-memory algorithms to maintain the core numbers of nodes in dynamic graphs. A parallel work of core decomposition for big graphs named WG_M is independently studied by [44]. WG_M also follows the semi-external model and processes each node sequentially. After computing the core number of each node v, WG_M uses a global array to save all neighbors $u \in N(v)$ once the core number of u is not less than that of v, i.e., $core(u) \geq core(v)$. In each iteration, WG_M only processes the nodes in the array. Unlike our proposed algorithm in Chapter 4, WG_M cannot guarantee that the core numbers of nodes marked in the array will necessarily update.

k-core is also studied in many specific graph structures, such as weighted graphs [36, 28], directed graphs [35], random graphs [53, 65, 59, 41] and uncertain graphs [11], etc. Locally computing and estimating core numbers is studied in [64].

The query based community search problem based on k-core model is studied in [76] and [24], where [24] proposes a local search method. [46] gives an influential community model based on k-core. The k-core algorithm is also used as a subroutine to solve other problems, such as finding approximation of densest subgraph and computing cliques with size k [45]. The k-core metric only requires

the minimum number of neighbors for each vertex in the graph to be no smaller than k. Therefore the number of non-neighbors for each vertex can be large. It is difficult to retain the familiarity when the size of a k-core is large.

Closely related to core decomposition, graph degeneracy is first defined in [49]. A greedy algorithm for computing the graph degeneracy order is proposed in [56]. Graph degeneracy can be applied in graph clustering [34, 18] and graph coloring [56]. Recently, [32] gives an approximate algorithm for computing the degeneracy order in large graphs.

Triangle-Based Metrics. k-truss has been investigated in [23, 80, 71]. It requires each edge in a k-truss is contained in at least $k-2$ triangles. k-truss has similar problem as k-core. It is easy to see that two cohesive subgraphs can be identified as one k-truss if they share mere one edge. In addition, k-truss is invalid in some popular graphs such as bipartite graphs. This model is also independently defined as k-mutual-friend subgraph in [96]. Based on triangles, DN-graph [83] with parameter k is a connected subgraph $G'(V', E')$ satisfying following two conditions: 1) Every connected pair of vertices in G' shares at least λ common neighbors. 2) For any $v \in V \backslash V', \lambda(V' \cup \{v\}) < \lambda$; and for any $v \in V', \lambda(V' \backslash \{v\}) \leq \lambda$. Such metric seems a little strict and generates many redundant results. Also, detecting all DN-graphs is NP-Complete. Approximate solutions are given in [83].

2.2.3 Connectivity Cohesiveness

In this category, most of existing works only consider edge connectivity of a graph. The edge connectivity of a graph is the minimum number of edges whose removal disconnect the graph. [91] first proposes algorithm to efficiently compute frequent closed k-edge connected subgraphs from a set of data graphs. However, a frequent closed subgraph may not be an induced subgraph. To conquer this

problem, [98] gives a cut-based method to compute all k-edge connected components in a graph. To further improve efficiency, [17] proposes a decomposition framework for the same problem and achieves a high speedup. The state-of-the-art solution for k-VCC enumeration is proposed in [48]. They provide a framework for correctly computing all k-VCCs. They also propose an approximate algorithm to achieve speedup. However, no any approximation ratio is given. [75] computes k-VCCs for small k values.

Vertex Connectivity. [31] proves the time complexity of computing maximum flow reaches $O(n^{0.5}m)$ in an unweighted directed graph while each vertex inside has either a single edge emanating from it or a single edge entering it. This result is used to test the vertex connectivity of a graph with given k in $O(n^{0.5}m^2)$ time. [30] further reduces the time complexity of such problem to $O(k^3m + knm)$. There are also other solutions for finding the vertex connectivity of a graph [33, 29]. To speed up the computation of vertex connectivity, [21] finds a sparse certificate of k-vertex connectivity. It is obtained by performing scan-first search k times.

Chapter 3

STRUCTURAL GRAPH CLUSTERING BASED COMMUNITY DETECTION

3.1 Overview

In this chapter, we introduce our proposed index-based solution for efficient structural graph clustering. The work is published in [85] and the rest of this chapter is organized as follows. Section 3.2 gives preliminary definitions and formally defines the problem. Section 3.3 reviews existing solutions. Section 3.4 introduces our index structure and proposes the query algorithm. Section 3.5 describes the algorithms for maintaining index when graph updates occur. Section 3.6 practically evaluates of our proposed algorithms and reports the experimental results. Section 3.7 summarizes the chapter.

3.2 Preliminary

We introduce basic definitions for the graph clustering method SCAN before stating the problem.

Definition 1. (Structural Neighborhood) *The structural neighborhood of a vertex u, denoted by $N[u]$, is defined as $N[u] = \{v \in V | (u, v) \in E\} \cup \{u\}$.*

Given a vertex u, we define the structural degree of u as the cardinality of $N[u]$, i.e., $deg[u] = |N[u]|$. Based on the concept of *structural neighborhood*, the *structural similarity* [90] is defined as follows.

Definition 2. (Structural Similarity) *The structural similarity between two vertices u and v, denoted by $\sigma(u, v)$, is defined as the number of common structural neighbors between u and v, normalized by the geometric mean of their cardinalities of the structural neighborhood. That is,*

$$\sigma(u, v) = \frac{|N[u] \cap N[v]|}{\sqrt{|N[u]||N[v]|}} \tag{3.1}$$

From the definition, we can see that the structural similarity between two vertices becomes large when they share many common structural neighbors. Intuitively, it is highly possible that two vertices belong to the same cluster if their structural similarity is large. In SCAN, a parameter ϵ is used as a threshold for the similarity value. Given a vertex u and a parameter ϵ ($0 < \epsilon \leq 1$), the *ϵ-neighborhood* [90] for u is defined as follows.

Definition 3. (*ϵ-neighborhood*) *The ϵ-neighborhood for a vertex u, denoted by $N_\epsilon[u]$, is defined as the subset of $N[u]$, in which every vertex v satisfies $\sigma(u, v) \geq \epsilon$. That is, $N_\epsilon[u] = \{v \in N[u] | \sigma(u, v) \geq \epsilon\}$.*

Note that the ϵ-neighborhood for a given vertex u includes u itself, since $\sigma(u, v) = 1$; when the number of ϵ-neighbors is large enough, we call u a *core*.

Note that in Chapter 4, we use the also term k-core to represent a degreee-based cohesive subgraph model. The term *core* used in this chapter only represents the concept in SCAN model. The formal definition of a *core* is given below. A parameter μ is used as the threshold for the cardinality of ϵ-neighborhood.

Definition 4. (Core) *Given a similarity threshold ϵ ($0 < \epsilon \leq 1$) and an integer μ ($\mu \geq 2$), a vertex u is a core if $|N_\epsilon[u]| \geq \mu$.*

A vertex is called a *non-core vertex* if it is not a core. Identifying cores is a crucial task in SCAN, given clusters can be obtained by expanding the cores. Specifically, each core u is considered as a seed vertex, and all of its ϵ-neighbors v belong to the same cluster of u, since u and v are similar enough. Once v is also a core, the seed scope will be expanded and all ϵ-neighbors of v will be added into the cluster as well. To describe such transitive relation, *structural reachability* [90] is defined as follows.

Definition 5. (Structural Reachability) *Given two vertices u and v, v is structurally reachable from u if there is a sequence of vertices $v_1, v_2, ...v_l \in V (l \geq 2)$ such that: (i) $v_1 = u, v_l = v$; (ii) for all $1 \leq i \leq l-1$, v_i is core, and $v_{i+1} \in N_\epsilon[v_i]$.*

A cluster is obtained when all structurally reachable vertices from any core vertex are identified. Below, we formally summarize the definition of a *cluster*.

Definition 6. (Cluster) *A cluster $C \in V$ is a non-empty subset of V such that:*

- (Connectivity) *For any two vertices $v_1, v_2 \in C$, there exists a vertex $u \in C$ such that both of v_1 and v_2 are structurally reachable from u.*
- (Maximality) *For a core $u \in C$, all vertices that are structurally reachable from u are also belong to C.*

Example 1. *We give an example of clustering result for the graph G in Fig. 3.1, where $\epsilon = 0.7, \mu = 4$. The structural similarity for every pair of adjacent vertices is given. For each pair of adjacent vertices, we represent the edge by a solid line if the structural similarity between them is not less than 0.7. Otherwise, we use a dashed line. We have four cores, namely v_0, v_2, v_9, and v_{10}. They are marked with the color black. All vertices that are structurally reachable from cores are marked in gray. We can see that there are two clusters obtained in the graph. In the cluster $\{v_0, v_1, v_2, v_3, v_4, v_5\}$, all inside vertices are structurally reachable from v_2 (connectivity). The vertices that are structurally reachable from v_0 and v_2 are all included in the cluster (maximality).*

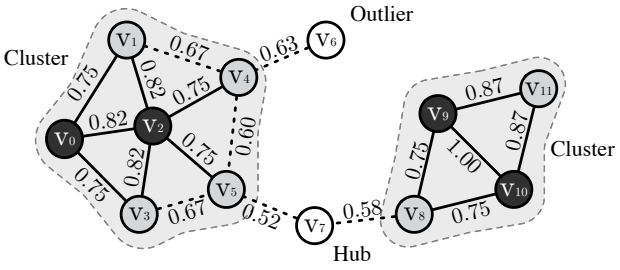


Figure 3.1: Clusters, hubs and outliers under $\epsilon = 0.7, \mu = 4$

Problem Statement. Given a graph $G(V, E)$ and two parameters $0 < \epsilon \leq 1$ and $\mu \geq 2$, we aim to efficiently compute the set $\mathbb{C}$ of all clusters in G.

Hub and Outlier. SCAN effectively identifies not only clusters but also *hubs* and *outliers*. The definition of *hub* and *outlier* in SCAN is given as follows.

Definition 7. (Hub and Outlier) *Given a vertex u that does not belong to any cluster, u is a hub if it has neighbors belonging to two or more different clusters. Otherwise, u is an outlier.*

Example 2. *In the case of Fig. 3.1, vertex v_7 is a hub, given it has two neighbors, namely v_5 and v_8, belonging to different clusters. Vertex v_6 is an outlier.*

In this chapter, we mainly focus on computing all clusters. Given the set of clusters in G, all hubs and outliers can be linearly obtained in $O(m+n)$ time, according to the definition.

3.3 Existing Solutions

In this section, we briefly review existing solutions. First, we introduce the original algorithm SCAN [90]. Then we present the state-of-the-art algorithm pSCAN [15].

3.3.1 SCAN

The original algorithm SCAN is proposed in [90]. For clearness of presentation, we give the pseudocode of SCAN in Algorithm 1, which is a version rearranged by [15] and is equivalent to the original one [90]. The pseudocode is self-explanatory.

Algorithm 1 SCAN [90]

Input: a graph $G(V, E)$ and parameters $0 < \epsilon \leq 1$ and $\mu \geq 2$
Output: the set $\mathbb{C}$ of clusters in G

1: **for each** edge $(u, v) \in E$ **do** compute $\sigma(u, v)$;
2: $\mathbb{C} \leftarrow \emptyset$;
3: **for each** unexplored vertices $u \in V$ **do**
4: $\quad C \leftarrow \{u\}$;
5: $\quad$ **for each** unexplored vertices $v \in C$ **do**
6: $\quad\quad$ mark v as explored;
7: $\quad\quad$ **if** $|N_\epsilon[v]| \geq \mu$ **then** $C \leftarrow C \cap N_\epsilon[v]$;
8: $\quad$ **if** $|C| > 1$ **then** $\mathbb{C} \leftarrow \mathbb{C} \cap \{C\}$;
9: **return** $\mathbb{C}$;

The total time complexity of SCAN is $O(\alpha \cdot m)$. Here α is the arboricity of G, which is the minimum number of spanning forests needed to cover all the edges of the graph G and $\alpha \leq \sqrt{m}$ [22]. In line 1, it costs $O(\alpha \cdot m)$ time to compute the structural similarity for each pair of adjacent vertices, which dominates the total

time complexity in the algorithm. Given all structural similarities, computing all clusters only needs $O(m)$ time.

3.3.2 pSCAN

Even though the algorithm SCAN is worst-case optimal [15], it requires a high computational cost to compute structural similarities. To handle this issue, [15] proposes a new algorithm called pSCAN, which is the state-of-the-art solution for this problem. The main idea of pSCAN is based on three observations: 1) The clusters may overlap; 2) The clusters of cores are disjointed; and 3) The clusters of non-core vertices are uniquely determined by cores.

pSCAN first clusters all cores, as every resulting cluster is uniquely identified by the cores inside it. Then, it assigns non-core vertices to corresponding clusters. The pseudocode is given in Algorithm 2.

Algorithm 2 pSCAN [15]

Input: a graph $G(V, E)$ and parameters $0 < \epsilon \leq 1$ and $\mu \geq 2$
Output: the set $\mathbb{C}$ of clusters in G

```
1: initialize a disjoint-set data structure with all u in V;
2: for each u ∈ V do
3:     sd(u) ← 0;
4:     ed(u) ← deg[u];
5: for each u ∈ V in non-increasing order w.r.t. ed(u) do
6:     CheckCore(u);
7:     if sd(u) ≥ μ then ClusterCore(u);
8: C_c ← the set of subsets of cores in the disjoint-set data structure;
9: ClusterNonCore();
10: return C;
```

To reduce unnecessary similarity computations, pSCAN maintains a lower bound $sd(u)$ and an upper bound $ed(u)$ of the number of ϵ-neighbors for each vertex u. Specifically, let $N'[u]$ be the set of neighbors of u such that the structural similarity between u and every $v \in N'[v]$ has been computed. $N'[u]$ is dy-

namically updated in the algorithm. Accordingly, $sd(u)$ and $ed(u)$ are assigned by $|\{v \in N'[u] | \sigma(u,v) \geq \epsilon\}|$ and $deg[u] - |\{v \in N'[u] | \sigma(u,v) < \epsilon\}|$ respectively. A vertex is a core if $sd(u) \geq \mu$ and is a non-core vertex if $ed(u) < \mu$. $sd(u)$ and $ed(u)$ are initialized by 0 and $deg[u]$ respectively (line 3 and line 4).

In line 6 of Algorithm 2, CheckCore(u) is invoked for confirming whether the given vertex u is a core, and for updating $sd(v)$ and $ed(v)$ for all unexplored neighbors $v \in N[u]$. In line 7, ClusterCore(u) assigns u and each $v \in N[u]$ to the same cluster based on the disjoint-set data structure if v is also a core and $\sigma(u,v) \geq \epsilon$. All clusters with only cores are obtained after line 7. Finally, in line 9, non-core vertices are assigned to corresponding clusters, where necessary.

Several optimization techniques for similarity-checking exist in pSCAN. Given two vertices, u and v, these techniques propose necessary conditions for both $\sigma(u,v) < \epsilon$ and $\sigma(u,v) \geq \epsilon$. Specifically, if $deg[u] < \epsilon^2 \cdot deg[v]$ *or* $deg[v] < \epsilon^2 \cdot deg[u]$, *then* $\sigma(u,v) < \epsilon$; and if $|N[u] \cap N[v]| \geq \lceil \epsilon \cdot \sqrt{deg[u] \cdot deg[v]} \rceil$, then $\sigma(u,v) \geq \epsilon$. This helps the algorithm achieve increased speed when structural similarity checking. In worst-case scenarios, the time complexity of pSCAN is still bounded by $O(\alpha \cdot m)$.

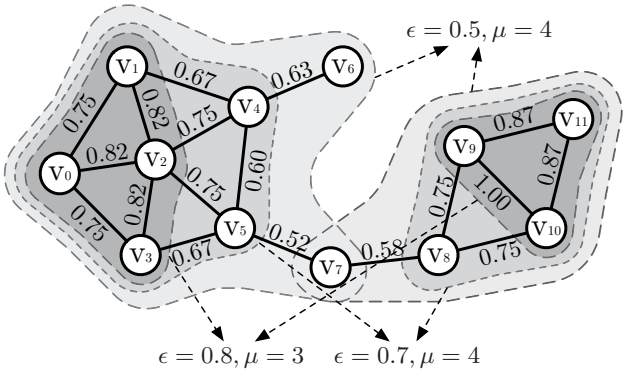


Figure 3.2: Clusters under different μ and ϵ

Drawbacks of Existing Solutions. All existing solutions focus on the online-

computing of all clusters via two exact given parameters. However, the change of input parameter value may heavily influence on the clustering result, especially in large graphs. We consider an example in Fig. 3.2. We use a dashed line to circle the clusters obtained by given ϵ and μ. The results may change even though we only slightly adjust ϵ or μ. Additionally, all existing methods always need to scan an entire graph to obtain its result clusters; for big graphs, this may consume a huge chunk of time.

Motivated by this, we propose an index-based method. With the index, we can answer the query for any given $0 < \epsilon \leq 1$ and $\mu \geq 2$ in a time complexity that is proportional to only the size of result subgraphs. To make our solution scalable to big graphs, the index size should be well bounded, and the time cost to index construction should be acceptable. Additionally, to handle the frequent updates in many real-world graphs, we also propose maintenance algorithms for our index structure. We discuss the details of index implementation in the following section.

3.4 Index-Based Algorithms

A Basic Index Structure. A straightforward idea for our index structure is the maintenance of structural similarity for each pair of adjacent vertices. We name such index by GS-Index. The construction for GS-Index is the same as in line 1 of Algorithm 1. Specifically, to calculate all structural similarities, we need to compute the number of common neighbors for each pair of adjacent vertices. This is equivalent to enumerating all triangles in the graph [22]. The space complexity of GS-Index, and time complexity of GS-Index construction, are summarized in the following lemmas.

Lemma 1. *The space complexity of* GS-Index *is* $O(m)$ *and the time complexity*

for constructing GS-Index *is* $O(\alpha \cdot m)$.

Lemma 2. *The time complexity of the query algorithm based on* GS-Index *is* $O(m)$.

Given all similarities, the result clusters are easily obtained by scanning the graph following the same procedures as detailed in lines 2–8 in Algorithm 1. We can see that even though the space usage of GS-Index can be well bounded, it still needs to traverse the entire graph to obtain the result clusters in query processing, and this may be hard to tolerate in big graphs. To address this issue, we propose a novel index structure, namely GS*-Index. With GS*-Index, we can query all result clusters in $O(m_c)$ time, where m_c is the number of edges in the induced subgraphs of the result clusters. We use $O(m)$ space and $O((\alpha + \log n) \cdot m)$ time to save and construct the index, respectively.

We provide an overview of our index structure in Subsection 3.4.1. In Subsection 3.4.2, we provide the implementation details for index construction. We propose the query algorithm in Subsection 3.4.3.

3.4.1 Index Overview

In this section, we introduce a novel index structure, namely GS*-Index. GS*-Index contains *core-orders* and *neighbor-orders* in addition to the structural similarity for each pair of adjacent vertices. Recall that different clusters may overlap while each core only belongs to a unique cluster. In this section, we first discuss the index for clustering cores. We will also show that our index can be naturally used to cluster together non-core vertices.

Core-Orders: Efficient Core Detection

The general idea underpinning our index is the maintenance of cores for every given ϵ and μ. In other words, with such index, we can efficiently obtain all cores by given any ϵ and μ. We address this problem by first computing all cores of any given similarity threshold ϵ under a specific μ. This is because that the parameter μ cannot be larger than the maximum structural degree, and there exists only a limited number of possible μ. Next, in the rest of this section, we discuss the candidate vertices nominated as cores within a given μ, and then we consider the precise obtainment of cores within a given ϵ in the candidate set.

The following observation presents the maximum and minimum possible value of input parameter μ.

Observation 1. *Given a graph G, we have $2 \leq \mu \leq d_{max}$, where $d_{max} = \max(deg[u]|u \in V)$.*

If $\mu = 1$, each vertex u would be an isolated result cluster, which is meaningless; there would be no result if $\mu = 0$ or if $\mu > d_{max}$. Additionally, in each given μ, not every vertex has the probability of being a core in a result cluster.

Observation 2. *Given a graph G and a parameters $2 \leq \mu \leq d_{max}$, vertex u cannot be a core in any cluster if $deg[u] < \mu$.*

According to Observation 1 and Observation 2, the set of all candidate vertices that will be cores is $\{u \in V|deg[u] \geq \mu\}$.

Now, given the candidate set for each specific μ, we consider detecting cores by similarity threshold ϵ. Recall that a vertex u is a core if there exist not less than μ ϵ-neighbors. We have the following lemma.

Lemma 3. *Given a graph G, a parameter $\mu \geq 2$, and two similarity thresholds $0 < \epsilon \leq \epsilon' \leq 1$, a vertex u is a core in a cluster obtained by μ and ϵ if it is a core in a cluster obtained by μ and ϵ'.*

According to the above lemma, we only need to save the maximum similarity value ϵ for each vertex u that will be a core under each specific μ. We call such a value the *core-threshold* and it is formally defined as follows.

Definition 8. (CORE-THRESHOLD) *Given a parameter μ, the core-threshold for a vertex u, denoted by $\mathcal{CT}_\mu[u]$, is the maximum value of ϵ such that u is a core in clusters obtained by $[\mu, \epsilon]$.*

Given parameters ϵ and μ, we know a vertex u is a core if $\mathcal{CT}_\mu[u] \geq \epsilon$ and u is not a core otherwise. Assume that we have $\mathcal{CT}_\mu[u]$ for all vertices u under every $2 \leq \mu \leq d_{max}$. For any given μ' and ϵ', to obtain all cores, a straightforward method is the checking of $\mathcal{CT}_{\mu'}[u]$ for all vertices u such that $deg[u] \geq \mu'$. However, this costs n times checking, even though there exists only a small number of cores. To reduce unnecessary checking, we give the following lemma.

Lemma 4. *Given a graph G and two parameters ϵ and μ, a vertex u is a core in result clusters if (i) v is a core; and $(ii)\mathcal{CT}_\mu[v] \leq \mathcal{CT}_\mu[u]$.*

According to Lemma 4, we know that if a given vertex u is a core, all vertices v with $\mathcal{CT}_\mu[v] \geq \mathcal{CT}_\mu[u]$ must be cores. To efficient obtain all cores, we sort the vertices in non-increasing order of their core-thresholds for each μ. We define such order as follows.

Definition 9. (CORE-ORDER) *The core-order for a given parameter μ, denoted by $\mathcal{CO}_\mu$, is a vertex order such that: (i) for all $v \in \mathcal{CO}_\mu, deg[u] \geq \mu$; and (ii) for any two vertices u and v, u appears before v if the core-threshold of u is not smaller than that of v, i.e., $\mathcal{CT}_\mu[u] \geq \mathcal{CT}_\mu[v]$.*

Given parameters μ and ϵ, all cores can be successfully obtained using the core-order. The identification starts from the first vertex in the core-order $\mathcal{CO}_\mu$

and terminates once there is a vertex whose core-threshold is less than the given similarity threshold ϵ.

We compute core-orders for all μ as a part of our index. The space complexity of all core-orders is given as follows:

Lemma 5. *The space cost of core-orders for all possible μ is bounded by $O(m)$.*

Neighbor-Orders:Efficient Cluster Construction

Core-Threshold Computation. A crucial task in core identification is computing the core-thresholds for each vertex under every possible μ. Based on the definition of cores, we propose following lemma.

Lemma 6. *Given a parameter μ, the core-threshold of a vertex u ($deg[u] \geq \mu$) is the μ-th largest value in structural similarities between u and its structural neighbors.*

According to Lemma 6, for each vertex u, we compute the structural similarities between u and all neighbors $v \in N[u]$, and sort the neighbors of u in a non-increasing order of their structural similarities. Consequently, the core-thresholds for all possible μ of u are obtained. We define such order as follows.

Definition 10. (Neighbor-Order) *The neighbor-order for a given vertex u, denoted by $\mathcal{NO}_u$, is a vertex order such that: (i) for all $v \in \mathcal{NO}_u, v \in N[u]$; and ($ii$) for any two vertices v_1 and v_2, v_1 appears before v_2 if the structural similarity between u and v_1 is not smaller than that between u and v_2, i.e., $\sigma(u, v_1) \geq \sigma(u, v_2)$.*

Based on Lemma 6 and Definiton 10, given a parameter μ, the core-threshold for a vertex u can be easily obtained by using the neighbor-order $\mathcal{NO}_u$. In addition to obtaining all core-thresholds for each vertex, we can also use neighbor-orders to construct clusters based on obtained cores.

Clusters Construction. Thanks to core-orders, we can efficiently obtain all cores. Then, to construct clusters from known cores u, we need to obtain all vertices v that are structurally reachable from u. Recall that the structural reachability transmits between cores only if they are ϵ-neighbor of each other. Thus, the construction of clusters can be solved by computing the ϵ-neighbors of each core.

All ϵ-neighbors of a given vertex u can be efficiently identified via the neighbor-order of u. Identification originates in the neighbor-order's first vertex, and terminates upon identification of a vertex whose structural similarity to u is smaller than the given similarity threshold ϵ. GS*-Index also computes neighbor-orders for all vertex u as a part of GS*-Index. There are $deg[u]$ items in the neighbor-order of each u, and the size of all neighbor-orders can be well bounded, as follows:

Lemma 7. *The space cost of neighbor-orders for all vertices in the graph is bounded by $O(m)$.*

Based on Lemma 5 and Lemma 7, we have:

Theorem 1. *The space cost of* GS*-Index *is bounded by $O(m)$.*

An Example of GS-Index*

We continue to use the graph G in Fig. 3.1 to give the example of our proposed GS*-Index.

Example 3. *The neighbor-order for each vertex in G is presented in Fig. 3.3. For each vertex u, the order is shown vertically and the structural similarity $\sigma(u, v)$ is presented over each neighbor v. Consider vertices v_3. There are four structural neighbors, namely v_3, v_0, v_2 and v_5. We sort the neighbors by*

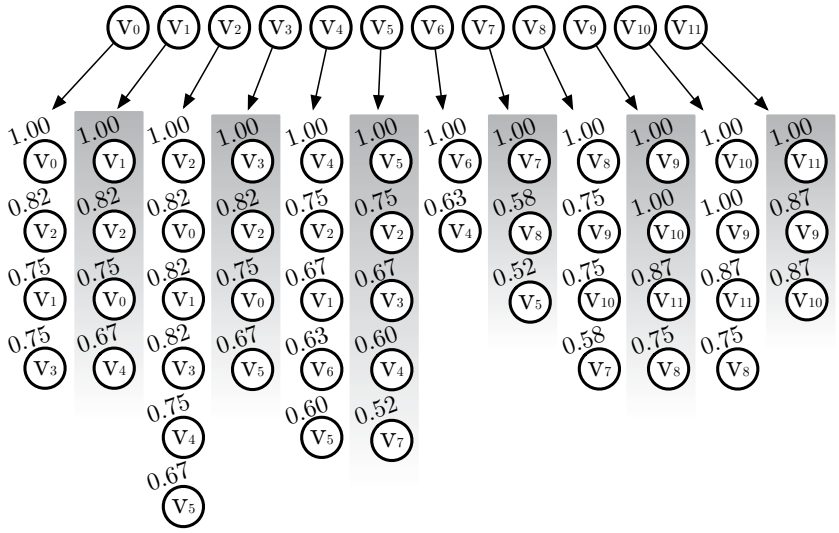


Figure 3.3: Neighbor-order for each vertex in graph G

their structural similarity to v_3*, and obtain the neighbor-order of* v_3*, which is* $[(1.00, v_3),(0.82, v_2),\ (0.75, v_0),(0.67, v_5)]$*. Given* $\epsilon = 0.7$*, to obtain* ϵ*-neighbors of vertex* v_3*, we iteratively check items in the neighbor-order of* v_3*. Its* ϵ*-neighbors are* v_3, v_2 *and* v_0*. The iteration terminates when checking neighbor* v_5*, since the structural similarity between* v_5 *and* v_3 *is less than* 0.7*.*

Given the neighbor-order for each vertex u*, we obtain the core-threshold of* u *under any given* μ*. Assume* $\mu = 3$*. The core-threshold of* v_3 *is the structural similarity between* v_3 *and the third vertex in the neighbor-order of* v_3*, which is* 0.75*. That means* v_3 *is a core if the other given parameter* ϵ *is not larger than* 0.75*, i.e.,* $\epsilon \leq 0.75$*. Otherwise,* v_3 *is not a core.*

Based on the neighbor-orders, we compute the core-orders for all possible μ*, and they are presented in Fig. 3.4. For each* μ*, the order is shown horizontally and the core-threshold* $\mathcal{CT}_u$ *is presented over* u*. Assume* $\epsilon = 0.7$ *and* $\mu = 4$*. To obtain all cores, we focus on the core-order for* $\mu = 4$ *and check vertices from left to right iteratively.* v_2, v_0, v_9 *and* v_{10} *are cores because their core-thresholds*

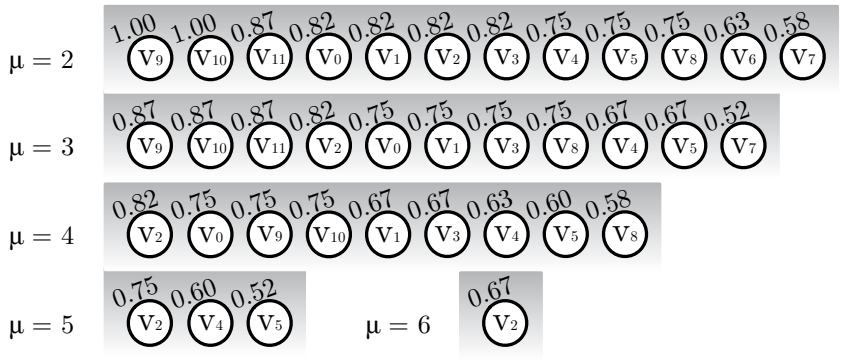


Figure 3.4: Core-order for each μ in graph G

are all not less than 0.7. *The iteration terminates when checking vertex* v_1, *as its core-threshold is less than* 0.7.

3.4.2 Index Construction

We present the index construction algorithm, which is named GS*-Construct, in this section. The pseudocode of GS*-Construct is given in Algorithm 3.

Algorithm 3 GS*-Construct(G)

Input: a graph $G(V, E)$
Output: GS*-Index of G

1: **for each** edge $(u, v) \in E$ **do** compute $\sigma(u, v)$;
2: $\mathcal{NO} \leftarrow \emptyset$;
3: **for each** vertex $u \in V$ **do**
4: $\quad \mathcal{NO}_u \leftarrow N[u]$;
5: $\quad$ sort vertices in $\mathcal{NO}_u$ according to Definiton 10;
6: $\quad \mathcal{NO} \leftarrow \mathcal{NO} \cup \{\mathcal{NO}_u\}$;
7: $\mathcal{CO} \leftarrow \emptyset$;
8: **for each** $\mu \in [2, d_{max}]$ **do**
9: $\quad \mathcal{CO}_\mu \leftarrow \{u \in V | deg[u] \geq \mu\}$;
10: $\quad$ sort vertices in $\mathcal{CO}_\mu$ according to Definiton 9;
11: $\quad \mathcal{CO} \leftarrow \mathcal{CO} \cup \{\mathcal{CO}_\mu\}$;
12: **return** $\mathcal{NO}$ and $\mathcal{CO}$;

Algorithm 3 computes the similarities for every pair of adjacent vertices in line 1, which is equivalent to counting all triangles in the graph. We adopt the same method in [15].

Neighbor-Order Computation. Algorithm 3 computes neighbor-orders for all vertices in lines 2-6. $\mathcal{NO}$ can be implemented as a two-dimensional array that saves the neighbor-order for each vertex. In each neighbor-order $\mathcal{NO}_u$, we sort the structural $v \in N[u]$ in a non-increasing order of $\sigma(u, v)$ (line 5).

Core-Order Computation. Algorithm 3 computes core-orders for all possible μ in lines 7-11. All vertices that have probability of being cores in the current μ are collected in line 9. Each item in $\mathcal{CO}_\mu$ is a vertex id. They are sorted by their core-thresholds under μ. Based on Lemma 6 and Definiton 10, the core-threshold $\mathcal{CT}_\mu[u]$ of a given vertex u is the structural similarity between u and μ-th items in $\mathcal{NO}_u$.

Theorem 2. *The time complexity of Algorithm 3 is* $O((\alpha + \log n) \cdot m)$.

Proof. The time complexity for computing all structural similarities (line 1) is $O(\alpha \cdot m)$, and has been discussed previously.

For each vertex u in lines 4–6, the time cost for sorting all neighbors is bounded by $O(deg[u] \cdot \log deg[u])$. Thus the time cost for all vertices is bounded by $O(\sum_{u \in V} deg[u] \cdot \log deg[u])$. Given $\sum_{u \in V} deg[u] = 2m$, we have $\sum_{u \in V} deg[u] \cdot \log deg[u] \leq 2m \cdot \log n$. The time complexity of line 2–6 is bounded by $O(m \cdot \log n)$.

For each μ in lines 9–11, it costs $O(|\mathcal{CO}_\mu| \log |\mathcal{CO}_\mu|)$ time to sort vertices in $\mathcal{CO}_\mu$. The time complexity for all μ in lines 7–11 is $O(\sum_{2 \leq \mu \leq d_{max}} |\mathcal{CO}_\mu| \log |\mathcal{CO}_\mu|)$. Each vertex u totally appears in $deg[u]$ arrays in $\mathcal{CO}$, resulting in $\sum_{2 \leq \mu \leq d_{max}} |\mathcal{CO}_\mu| = 2m$. $|\mathcal{CO}_\mu|$ gradually decreases when increasing μ and $\mathcal{CO}_2 = n$. We have $|\mathcal{CO}_\mu| \leq n$ for any μ. Thus the time complexity of lines 7–11 can be bounded by $O(m \cdot \log n)$.

Summing the time complexity of all three parts above, total time complexity of Algorithm 3 is $O((\alpha + \log n) \cdot m)$. □

Normally, the arboricity α is much greater than $\log n$ especially for big graphs. Thus, we can bound the total time complexity of Algorithm 3 by $O(\alpha \cdot m)$, which is same as that for pSCAN in the worst case.

3.4.3 Query Processing

In this section, we discuss the query processing procedure, named GS*-Query, which is based on our proposed index GS*-Index. The general idea of GS*-Query is iteratively finding an unexplored core u; then computing all vertices v that are structurally reachable from u, and grouping them into a result cluster. Before introducing the details of GS*-Query, we give the following observation based on *Definiton* 6.

Observation 3. *Given a cluster C and any core $u \in C$, all vertices $v \in C$ are structural reachable from u.*

From Observation 3, we know that the cluster obtained from any inside core is complete. The detailed pseudocode is given in Algorithm 4. Note that GS*-Query no longer needs the original graph G as an input parameter, given the neighborhood information can be obtained via the neighbor-order for each vertex.

To obtain all clusters, Algorithm 4 iteratively processes vertices according the order in $\mathcal{CO}_\mu$. It performs until we find a vertex u whose core-threshold is less than given ϵ (line 4).

Algorithm 4 computes the cluster C containing a core u from line 5–20. This is done by using a queue. For each vertex v in the queue, we add all ϵ-neighbors of v into the cluster and add new discovered cores into the queue (line 11–20). Specifically, we iteratively process each neighbor w of v according to

their position in $\mathcal{NO}_v$ (line 10). The iteration terminates once we find a vertex w that is not ϵ-neighbor of v (line 12). The explored neighbors are skipped, as they have been added into the cluster previously (line 13). We add the unexplored neighbors into the cluster in line 15. In line 17, we identify whether w is a core by checking the core-threshold of w, which is the μ-th structural similarity in $\mathcal{NO}_w$. We add w to the queue to detect more structurally reachable vertices if w is a core in line 18. The cluster containing u is added to the result set in line 21. In line 22 we mark all non-core vertices back to unexplored before computing new clusters, given the non-core vertices may overlap between different clusters. A running example of Algorithm 4 is given below.

Algorithm 4 GS*-Query

Input: GS*-Index and parameters $0 < \epsilon \leq 1, \mu \geq 2$;
Output: the set $\mathbb{C}$ of clusters in G;

```
1: ℂ ← ∅;
2: for each u ∈ 𝒞𝒪_μ do
3:    v ← μ-th vertex in 𝒩𝒪_u;
4:    if σ(u, v) < ε then break;
5:    C ← {u};
6:    𝒬 ← initialize an empty queue;
7:    𝒬.insert(u);
8:    mark u as explored;
9:    while 𝒬 ≠ ∅ do
10:      v ← 𝒬.pop();
11:      for each w ∈ 𝒩𝒪_v do
12:         if σ(v, w) < ε then break;
13:         if w is explored then continue;
14:         mark w as explored;
15:         C ← C ∪ {w};
16:         t ← μ-th vertex in 𝒩𝒪_w;
17:         if σ(w, t) ≥ ε then
18:            𝒬.insert(w);
19:         else
20:            mark w as non-core vertex;
21:   ℂ ← ℂ ∪ {C};
22:   for each non-core vertex v do mark v as unexplored;
23: return ℂ;
```

Example 4. *We give an example of Algorithm 4 for* $\epsilon = 0.7$, $\mu = 4$ *in Fig. 3.5. All neighbor-orders and core-orders for* G *in Fig. 3.1 can be found in Fig. 3.3 and Fig. 3.4, respectively. Given* $\mu = 4$, *Algorithm 4 first focuses on the core-order* $\mathcal{CO}_4$, *which is shown in the top of Fig. 3.5. Then, vertices in* $\mathcal{CO}_4$ *are processed iteratively. We mark in the color black the vertices that are inserted into the queue (line 7 and line 18 of Algorithm 4).*

Vertex v_2 *is the first vertex inserted into the queue, given its core-threshold is larger than* 0.7. *Algorithm 4 then assesses neighbors of* v_2 *following the neighbor-order* $\mathcal{NO}_{v_2}$. *The neighbor-order for each vertex in queue is shown around clock-wise below the core-order in Fig. 3.5.* v_2 *has been explored and thus is skipped. Then,* v_0 *is added to the cluster and inserted into the queue, as its core-threshold is* 0.75. *After that, vertices* $v_1, v_3 and v_4$ *are added to the cluster. Given none of them is core, they are not inserted into the queue. The iteration terminates after checking* v_5, *given the structural-similarity between* v_5 *and* v_2 *is less than* 0.7. *Algorithm 4 continues to process* v_0 *in the queue, and adds all its* ϵ*-neighbors to the cluster. A cluster is successfully obtained, since no vertex exists in the queue any more.*

In the core-order, vertex v_0 *is explored, and thus skipped. Algorithm 4 inserts* v_9 *into the queue and obtains the second cluster. Algorithm 4 terminates when checking* v_1 *in the core-order, since its core-threshold is less than* 0.7. *That means* v_1 *and all following vertices can not be cores.*

Theorem 3. *Given parameters* $\mu \geq 2$ *and* $0 < \epsilon \leq 1$, *Algorithm 4 correctly computes all clusters of the graph* G.

Proof. (Correctness) We first prove the correctness for each obtained cluster. Based on Lemma 6 and Definiton 8, Algorithm 4 successfully identifies cores by comparing given ϵ with the μ-th largest similarity value between a given vertex

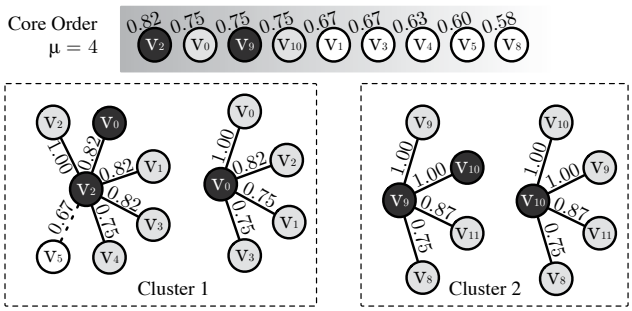


Figure 3.5: A running example for $\epsilon = 0.7, \mu = 4$

and its structural neighbors (line 4 and line 17). For each core v, the neighbors w are sorted in the non-increasing order of their structural similarities to v in $\mathcal{NO}_v$. Following this order, Algorithm 4 successfully obtains all ϵ-neighbors of v and inserts their cores into the queue. Then it traverses the unexplored ϵ-neighbors of the core in the queue until the queue is empty. That means all vertices that are structurally reachable from the original u in line 3 are obtained. According to Observation 3, we obtained the correct and complete cluster containing u. Note that all non-core vertices are marked back to unexplored. This guarantees we do not lose any non-core vertices in the following cluster computation.

(Completeness) Next, we prove the completeness of the resulting cluster set. Recall that given a parameter μ, vertices are sorted in the non-increasing order of their core threshold under μ in $\mathcal{CO}_\mu$. Based on Definiton 8, Algorithm 4 correctly finds all cores by following this order, and terminates once a non-core vertex is found (line 4). Since a cluster can be correctly obtained by an inside core, all clusters are successfully obtained in Algorithm 4.

(Redundancy Free) Since each core only belongs to a unique cluster, Algorithm 4 only loads ϵ-neighbors for each core, and marks all explored cores. This guarantees no repeated cluster exists in the result set. □

Let E_C be the set of edges in the induced subgraph of cluster C. The time

complexity of Algorithm 4 is given as follows.

Theorem 4. *The time complexity of Algorithm 4 is bounded by* $O(\sum_{C \in \mathbb{C}} |E_C|)$.

Proof. Let C_{core} be the set of cores in a cluster C. The total number of visited vertices in line 2 is $\sum_{C \in \mathbb{C}} |C_{core}| + 1$. Since all cores that belong to the same cluster are processed in lines 5–20, the total number of explored vertices found in line 2 is $|\mathbb{C}|$.

To obtain each cluster C, only cores are inserted into the queue. Thus, the number of iterations in the while loop (line 9) is $|C_{core}|$. For each core v in line 10, all of its ϵ-neighbors are visited and the number of iterations in the for loop (line 11) is $deg_C[v] + 1$, where $deg_C[v]$ is the degree of v in the induced subgraph of cluster C containing v. Thus the time complexity for computing a cluster (lines 5–20) is $O(\sum_{u \in C_{core}} deg_C[u])$. This can be also represented by $O(|E_C|)$. The total complexity of Algorithm 4 is $O(\sum_{C \in \mathbb{C}} |E_C|)$. □

Local Cluster Query. Our query processing algorithm can also be naturally extended to compute result clusters containing a query vertex u. Given a query vertex u and two parameters $\mu \geq 2$ and $0 < \epsilon \leq 1$, if u is a core, the algorithm is the same as in lines 5–20 of Algorithm 4.

Otherwise, u may be contained in several clusters. In this case, we process ϵ-neighbors v of u by following the order in $\mathcal{NO}_u$. If v is a core, we obtained the cluster containing u and v by similarly invoking lines 5–20 of Algorithm 4. We skip v if it is not a core.

Let $\mathbb{C}$ be the result set of clusters containing query vertex u. The time complexity for each of the two cases detailed above is also bounded by $O(\sum_{C \in \mathbb{C}} |E_C|)$. Note that for the case in which the query vertex is a core, only one cluster C exists in the result set $\mathbb{C}$.

3.5 Index Maintenance

In the previous section, we discussed our index-based solution for the graph clustering, namely GS*-Query. Compared to existing online-computing solutions, GS*-Query can correctly and efficiently compute the clusters by given any given parameters μ and ϵ. However, most of real-word graphs are frequently updated. A fixed and outdated index structure can hardly meet the expectations of efficiently clustering large, dynamic graphs. Motivated by this issue, in this section we discuss the algorithms for maintaining our proposed index structure when graphs update. Note that we mainly focus on the edge insertion and deletion, as the vertex updates can be handled by performing several edge updates.

3.5.1 Basic Solutions

In this section we give a basic solution for index maintenance. For the ease of presentation, we mainly discuss edge insertion. The ideas can be easily extended to handle edge removal.

Edge Insertion. Recall that our index structure GS*-Index contains two parts: neighbor-orders and core-orders. We give the following observation to reveal the relationship between them.

Observation 4. *The core-order $\mathcal{CO}_\mu$ for parameter μ does not change if the neighbor-order $\mathcal{NO}_u$ for each vertex $u \in \mathcal{CO}_\mu$ does not change.*

According to Observation 4, when inserting an edge, we first update the neighbor-orders of the corresponding vertices, and then update the core-orders, if necessary. In the neighbor-order for vertex u, the structural similarity between u and each neighbor $v \in N[u]$ is computed in accordance with their common neighbors and degrees. We have the following observation.

Algorithm 5 GS*-Insert

Input: an edge (u, v);
Output: updated index structure;

```
1: insert the edge(u, v) into graph G;
2: deg[u] ← deg[u] + 1, deg[v] ← deg[v] + 1;
3: UpdateVertex(u);
4: UpdateVertex(v);
5: Procedure UpdateVertex(u) :
6: NO_u ← ∅;
7: for each vertex w ∈ N(u) do
8:    recompute σ(u, w);
9:    if u ∈ NO_w then
10:      update u in NO_w according to σ(u, w);
11:   else
12:      insert u into NO_w according to σ(u, w);
13:   UpdateCores(w);
14:   insert w into NO_u;
15: UpdateCores(u);
16: Procedure UpdateCores(u) :
17: i ← 0;
18: for each v ∈ NO_u do
19:   i ← i + 1
20:   if v = u then continue;
21:   if u ∈ CO_i then
22:      update u in CO_i according to σ(u, v);
23:   else
24:      insert u into CO_i according to σ(u, v);
```

Observation 5. *The neighbor-order of vertex w changes if $\exists u \in N[w]$, and an edge (u, v) inserts into or is removed from u.*

Let (u, v) be an edge inserting into graph G. Following the above observation, we only need to update the neighbor-orders for the following vertices: $u, v, w \in N[u]$, and $w' \in N[v]$. Then we update the position of these vertices in the core-order for each parameter μ, based on their new core-thresholds.

The algorithm for edge insertion, namely GS*-Insert, is given in Algorithm 5. In Algorithm 5, we first physically insert u and v into the neighbor list of each

other in line 1, and update their degree in line 2. For the ease of presentation, we call u and v root vertices.

We invoke procedure UpdateVertex for root vertices. To maintain neighbor-orders, we first compute the new structural similarity between u and each $v \in N[u]$ (line 8). When the new structural similarity $\sigma(u, w)$ is found, we correspondingly update the neighbor-order for v correspondingly (line 10). Note that we check the existence for v because when inserting a new edge (u, v), vertex v does not exist in the neighbor-order of u. Since the neighbor-order for w changes, we update the corresponding core order by invoking UpdateCores(w). Given all items in the neighbor-order of root vertex u change, we update the core-orders influenced by u (line 15) after constructing an entire neighbor-order for u.

The procedural details for UpdateCores are also given in Algorithm 5. The first vertex is skipped, since it is the vertex itself (line 20). We update the position of vertex u in each core-order $\mathcal{CO}_i$ by the i-th structural similarity in $\mathcal{NO}_u$. In cases where u is a root vertex, u does not exist in the core-order $\mathcal{CO}_{deg[u]}$. Thus, we check the existence for vertex u in line 21. Vertex u is inserted into $\mathcal{CO}_{i+1}$ (line 24) if it does not exist.

Given a vertex u, let $\mathbb{N}[u]$ be the set of vertices whose distances to v are not larger than 2, and $E_{\mathbb{N}[u]}$ be the set of edges in the induced subgraph of $\mathbb{N}[u]$.

Theorem 5. *Given an inserted edge (u, v), the time complexity of Algorithm 5 is bounded by $O((|E_{\mathbb{N}[u]}| + |E_{\mathbb{N}[v]}|) \cdot \log n)$.*

Proof. In Algorithm 5, line 1 and line 2 cost constant time. The dominating cost is the invocation of UpdateVertex. For each neighbor w in line 7 of Algorithm 5, computing new structural similarity $\sigma(u, w)$ needs $O(deg[w])$ of time. This can be done by using a bitmap to mark all neighbors of u in advance. The algorithm updates the neighbor-order of w in lines 9–12. The update (line 10) or insertion (line 12) can be finished in $O(\log deg[w])$ time in the worst-case scenarios via

a self-balancing binary search tree that maintains the order. Similarly, we use such data structures to implement the core-orders. UpdateCores is invoked for a vertex w in line 13. We know that the size of any core-order is not larger than n. Thus the time complexity of line 8 is bounded by $O(deg[w] \cdot \log n)$. In line 14, it costs $O(\log deg[u])$ of time to insert a new vertex w into the neighbor-order of u. Summarizing the above time cost: the time complexity of UpdateVertex is $O(\sum_{w \in N[u]} deg[w] \cdot \log n)$, which can be also represented as $O(|E_{\mathbb{N}[u]}| \cdot \log n)$. The total time complexity of GS*-Insert can be easily bounded by $O((|E_{\mathbb{N}[u]}| + |E_{\mathbb{N}[v]}|) \cdot \log n)$. □

Algorithm 6 GS*-Remove

Input: an edge (u, v);
Output: updated index structure;

1: remove the edge(u, v) from graph G;
2: $deg[u] \leftarrow deg[u] - 1, deg[v] \leftarrow deg[v] - 1$;
3: remove u from order $\mathcal{CO}_{deg[u]}$;
4: UpdateVertex(u);
5: remove v from order $\mathcal{CO}_{deg[v]}$;
6: UpdateVertex(v);

Edge Removal. Following the similar idea for handling edge insertion, the procedure for edge removal is given in Algorithm 6. We invoke the same subroutine UpdateVertex for each vertex. A difference here is that we need to remove the root-vertex from the highest core-order in advance (line 3 and line 5).

Theorem 6. *Given a removed edge (u, v), the time complexity of Algorithm 6 is bounded by $O((|E_{\mathbb{N}[u]}| + |E_{\mathbb{N}[v]}|) \cdot \log n)$.*

Proof. The proof is similarity to that for Theorem 5, and thus is purposefully omitted here. □

3.5.2 Improved Algorithms

In the previous section, we gave a basic solution for maintaining our proposed index. In the procedure UpdateVertex of Algorithm 5, the dominating cost is the recomputing of the structural similarity for each neighbor w in line 8, and updating $deg[w]$ core-orders in line 13. In this section, we discuss several optimizations for reducing the time cost of these two tasks and propose optimized algorithms for the index maintenance.

Avoiding Structural Similarity Recomputation. The key to computing the structural similarity is calculating the number of common neighbors. Our general idea of avoiding recomputing structural similarity is to efficiently maintain the number of common neighbors for each adjacent pair of vertices. Note that storing the number of common neighbors for each pair of adjacent vertices needs $O(m)$ space, and thus the total space complexity can be still bounded by $O(m)$.

Recall that when inserting or removing an edge (u, v), only the neighbor-orders of structural neighbors of vertex u and v are influenced. We categorize vertices into the following three types:

- (Type I) the neighbors of either u or v, i.e., $N(u) \cup N(v) - N(u) \cap N(v)$;
- (Type II) the common neighbors of u and v, i.e., $N(u) \cap N(v)$;
- (Type III) the edge's terminals, i.e., u and v.

Considering an inserting or removing edge (u, v), u is an type III vertex. For each neighbor w of u, let $\mathcal{CN}_{old}(w, u)$ be the original number of common neighbors between w and u, and $\mathcal{CN}_{new}(w, u)$ be the updated value. We have the following observations.

Observation 6. *Given an inserted edge (u, v), for an open neighbor $w \in N(u)$, $\mathcal{CN}_{new}(w, u) = \mathcal{CN}_{old}(w, u)$ if w is a type I vertex, and $\mathcal{CN}_{new}(w, u) = \mathcal{CN}_{old}(w, u)+1$ if w is a type II vertex.*

Observation 7. *Given an removed edge (u, v), for an open neighbor $w \in N(u)$, $\mathcal{CN}_{new}(w, u) = \mathcal{CN}_{old}(w, u)$ if w is a type I vertex, and $\mathcal{CN}_{new}(w, u) = \mathcal{CN}_{old}(w, u) - 1$ if w is a type II vertex.*

Observation 6 and Observation 7 shows that we can efficiently obtain the new number of common neighbors for type I and type II vertices, and consequently obtain the new structural similarities.

In our basic solution (Algorithm 5), we compute the number of common neighbors between two type III vertices. Note that the type II vertices essentially are their open common neighbors; thus we can count the number of type II neighbors and avoid recomputing the number of common neighbors between two type III vertices. Details are shown in the pseudocodes.

Accurate Core-Order Updates. Besides avoiding naively recomputing structural similarities, we can also achieve increased speed in core-orders maintenance. Recall that the position of vertex u in the core-order $\mathcal{CO}_\mu$ may change only if its core threshold for μ (the μ-th structural similarity in its neighbor-order $\mathcal{NO}_u$) changes. Given a vertex u and a vertex $v \in N[u]$, assume that structural similarity $\sigma(u, v)$ between u and v changes from sim_{old} to sim_{new}. Let sim_l and sim_r be the smaller value and the larger value respectively in sim_{old} and sim_{new}. We have the following observation.

Observation 8. *Given a parameter μ, the core-threshold for vertex u changes if the μ-th structural similarity value in $\mathcal{NO}_u$ falls in range $[sim_l, sim_r]$.*

Based on Observation 8, we avoid the operation for updating the core-order if the corresponding core-threshold does not fall into the specified range.

We name the optimized insertion algorithm by GS*-Insert*. The pseudocode for GS*-Insert* is given in Algorithm 7. GS*-Insert* invokes UpdateAdd for each vertex. In the procedure UpdateAdd, *cn* is used to count the common neighbors

Algorithm 7 GS*-Insert*

Input: an edge (u, v);
Output: updated index structure;

1: insert the edge(u, v) into graph G;
2: $deg[u] \leftarrow deg[u] + 1, deg[v] \leftarrow deg[v] + 1$;
3: UpdateAdd(u);
4: UpdateAdd(v);
5: **Procedure** UpdateAdd(u, v) :
6: $\mathcal{NO}_u \leftarrow \emptyset$;
7: $cn \leftarrow 2$;
8: **for each** vertex $w \in N(u)$ **do**
9: **if** $w = v$ **then continue**;
10: $\sigma_{old}(u, w) \leftarrow$ existing structural similarity between u and w;
11: **if** $w \in N(v)$ **then**
12: $cn \leftarrow cn + 1$;
13: $\mathcal{CN}(u, w) \leftarrow \mathcal{CN}(u, w) + 1$;
14: $\sigma(u, w) \leftarrow \mathcal{CN}(u, w)/\sqrt{deg[u] \cdot deg[w]}$;
15: update u in $\mathcal{NO}_w$ according to $\sigma(u, w)$;
16: UpdateCores*$(w, \sigma_{old}(u, w), \sigma(u, w))$;
17: insert w into $\mathcal{NO}_u$ according to $\sigma(u, w)$;
18: $\mathcal{CN}(u, v) \leftarrow cn$;
19: $\sigma(u, v) \leftarrow \mathcal{CN}(u, v)/\sqrt{deg[u] \cdot deg[v]}$;
20: insert v into $\mathcal{NO}_u$ according to $\sigma(u, v)$;
21: UpdateCores*$(u, 1, 0)$;
22: **Procedure** UpdateCores*$(u, \epsilon_l, \epsilon_r)$:
23: **if** $\epsilon_r > \epsilon_l$ **then** $swap(\epsilon_r, \epsilon_l)$;
24: $i \leftarrow 0$;
25: **for each** $v \in \mathcal{NO}_u$ **do**
26: $i \leftarrow i + 1$;
27: **if** $\sigma(u, v) > \epsilon_l$ **then continue**;
28: **if** $\sigma(u, v) < \epsilon_r$ **then break**;
29: **if** $u \in \mathcal{CO}_i$ **then**
30: update u in $\mathcal{CO}_i$ according to $\sigma(u, v)$;
31: **else**
32: insert u into $\mathcal{CO}_i$ according to $\sigma(u, v)$;

between two type III vertices u and v. We check whether w is a type II vertex in line 11. If so, we increase cn and update the number of common neighbors between w and u according to Observation 6. We invoke an optimized procedure

UpdateCores* to update core-orders. In the procedure UpdateCores*, we only update core-orders if the structural similarity falls into the range $[\epsilon_r, \epsilon_l]$ (line 27–28).

Algorithm 8 GS*-Remove*

Input: an edge (u, v);
Output: updated index structure;

1: remove the edge(u, v) from graph G;
2: $deg[u] \leftarrow deg[u] - 1, deg[v] \leftarrow deg[v] - 1$;
3: UpdateDel(u);
4: UpdateDel(v);
5: **Procedure** UpdateDel(u) :
6: $\mathcal{NO}_u \leftarrow \emptyset$;
7: remove u from order $\mathcal{CO}_{deg[u]}$;
8: **for each** vertex $w \in N(u)$ **do**
9: $\sigma_{old}(u, w) \leftarrow$ existing similarity between u and w;
10: **if** $w \in N(v)$ **then**
11: $\mathcal{CN}(u, w) \leftarrow \mathcal{CN}(u, w) - 1$;
12: $\sigma(u, w) \leftarrow \mathcal{CN}(u, w)/\sqrt{deg[u] \cdot deg[w]}$;
13: update u in $\mathcal{NO}_w$ according to $\sigma(u, w)$;
14: UpdateCores*$(w, \sigma_{old}(u, w), \sigma(u, w))$;
15: insert w into $\mathcal{NO}_u$ according to $\sigma(u, w)$;
16: UpdateCores*$(u, 1, 0)$;

The pseudocode of our optimized edge removal algorithm, namely GS*-Remove*, is given in Algorithm 8, and it invokes UpdateDel for each vertex, which is similar to UpdateAdd. We empty the neighbor order of u in line 6 and remove u from the core order $\mathcal{CO}_{deg[u]}$ due to the degree drop of u. We check whether w is the common neighbor of u and v. Based on Observation 7, we decrease the common neighbor between u and w in line 11 when w is a type II vertex. The new structural structural similarity between u and w is computed in line 12. We update the position of u in the neighbor order of w in line 13, and invoke UpdateCores* to adjust the position w in the corresponding core-orders in line 14. We update the position of vertex u in every possible core-order in line 16.

3.6 Performance Studies

We conduct extensive experiments to evaluate the performance of our proposed solutions. Algorithms that appeared in the experiments are summarized as follows:

- pSCAN: The state-of-the-art algorithm for structural clustering in [15].
- GS-Construct: The algorithm that constructs the basic GS-Index, which directly computes structural similarity between every pair of adjacent vertices.
- GS*-Construct: The algorithm that constructs our proposed index GS*-Index in Section 3.4.
- GS-Query: The query algorithm based on GS-Index.
- GS*-Query: The query algorithm based on GS*-Index.
- GS*-Insert: The naive algorithm for edge insertion.
- GS*-Insert*: The optimized algorithm for edge insertion.
- GS*-Remove: The naive algorithm for edge removal.
- GS*-Remove*: The optimized algorithm for edge removal.

All algorithms are implemented in C++ and compiled using g++ complier at -O3 optimization level. All the experiments are conducted on a Linux operating system running on a machine with an Intel Xeon 2.8GHz CPU, 256GB 1866MHz DDR3-RAM. The time cost for algorithms is measured as the amount of wall-clock time elapsed during the program execution.

Datasets	$\|V\|$	$\|E\|$	$\overline{d}$	c
DBLP	317,080	10,498,66	6.62	0.6324
Amazon	403,394	3,387,388	16.79	0.4177
WIKI	2,394,385	5,021,410	4.19	0.0526
Google	875,713	5,105,039	11.66	0.5143
Cit	3,774,768	16,518,948	8.75	0.0757
Pokec	1,632,803	30,622,564	37.51	0.1094
LiveJournal	3,997,962	34,681,189	17.35	0.2843
Orkut	3,072,441	117,185,083	76.28	0.1666
UK-2002	18,520,486	298,113,762	32.19	0.6891
Webbase	118,142,155	1,019,903,190	17.27	0.5533

Table 3.1: Network statistics

Datasets. We use 10 publicly available real-world networks to evaluate the algorithms. The detailed statistics are shown in Table 3.1, where the average degree ($\overline{d}$) and average clustering coefficient (c) are shown in the last two columns. The networks are displayed in non-decreasing order regarding the number of edges. All networks and corresponding detailed description can be found in SNAP[1] and Webgraph[2].

3.6.1 Performance of Query Processing

We compare our best query algorithm GS*-Query with GS-Query and the state-of-the-art algorithm pSCAN for structural clustering in this section.

We first report the performance of the algorithms by varying μ and ϵ. For parameter settings, we adopt a similar settings that are used in [15] and [74]. For $0 < \epsilon \leq 1$, we choose $0.2, 0.4, 0.6$, and 0.8, with 0.6 as default and for $\mu \geq 2$, we choose $2, 5, 10$, and 15, with 5 as default. Because of space limitations, we only show the charts for WIKI, Pokec, Orkut and UK-2002, and note that the

[1] `http://snap.stanford.edu/index.html`
[2] `http://webgraph.di.unimi.it/`

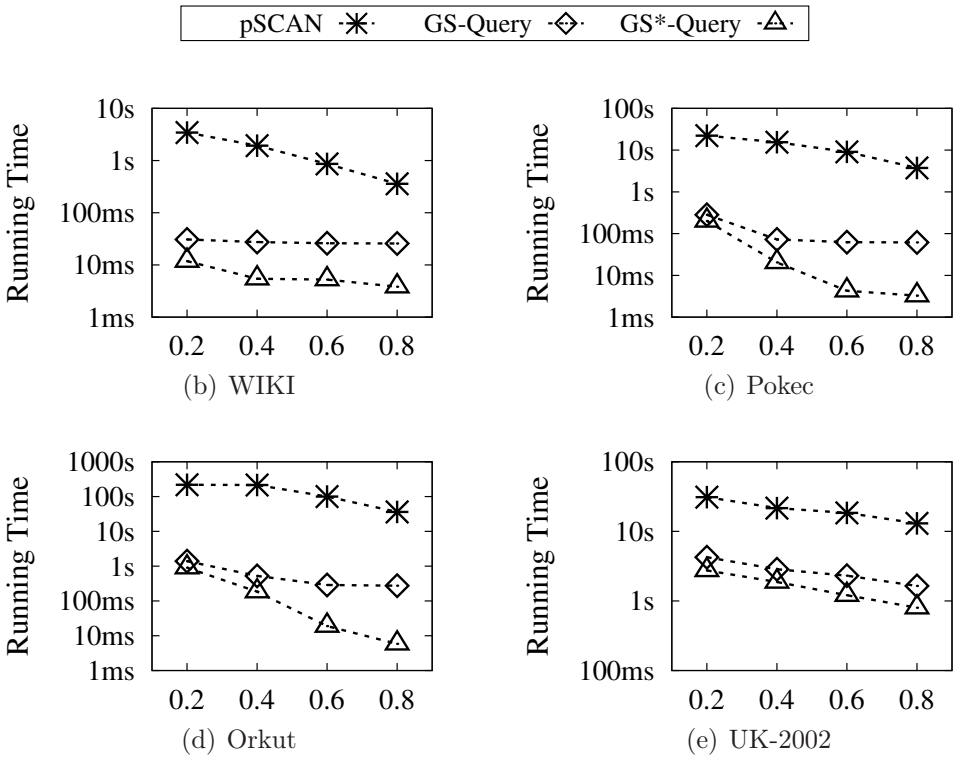


Figure 3.6: Query time for different ϵ ($\mu = 5$)

results of the other datasets present similar trends. The results for all datasets under default parameters settings are reported later.

Eval-I: Varying ϵ. The time cost of GS*-Query, GS-Query, and pSCAN by varying ϵ is given in Fig. 3.6. We can see that GS*-Query is faster than GS-Query under every ϵ on all datasets, and the gap between them gradually increases when ϵ grows. For example, in Orkut, GS*-Query spends about 0.9 seconds while GS-Query spends about 1.4 seconds when ϵ is 0.2. When ϵ reaches 0.8, it costs only 5 milliseconds to perform GS*-Query, while GS-Query still costs almost 0.3 seconds. Additionally, when ϵ grows, lines for GS*-Query present a significant downward trend on all datasets. This is because the time cost of GS*-Query is strictly dependent on the result size. The result size becomes smaller when

ϵ increases. The lines for GS-Query perform relatively steadier compared with GS*-Query. Lines for pSCAN on some datasets also present slightly downward trends, reflecting the pruning techniques used. However, GS*-Query is significantly faster than pSCAN especially when ϵ is large.

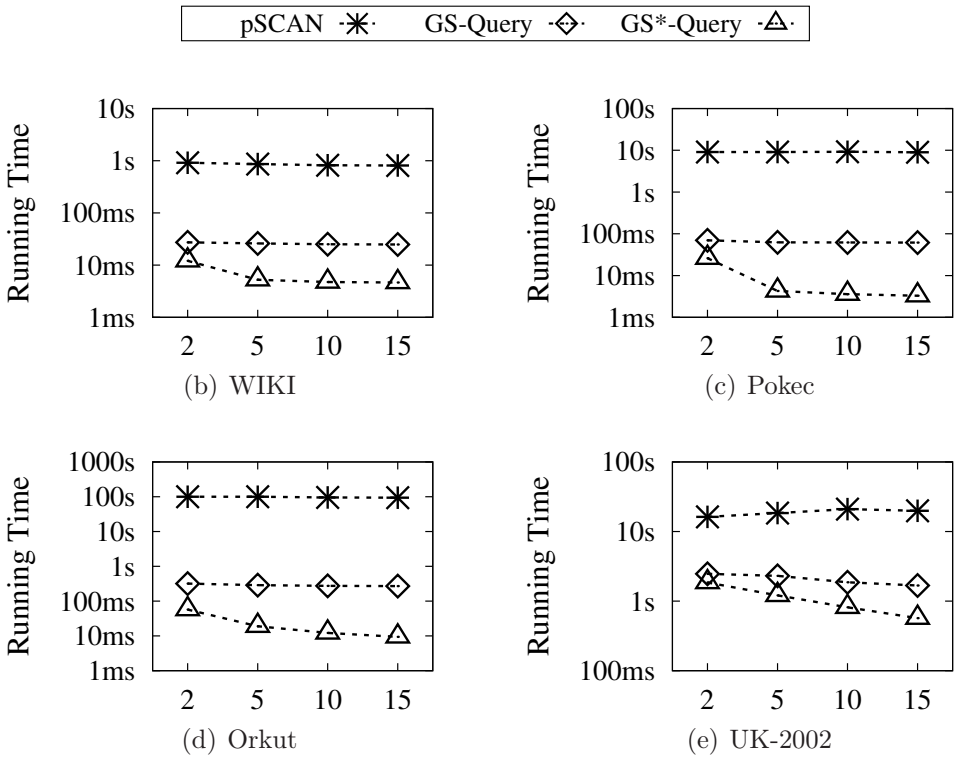


Figure 3.7: Query time for different μ ($\epsilon = 0.6$)

Eval-II: Varying μ. The time cost for GS*-Query, GS-Query and pSCAN by varying μ is given in Fig. 3.7. Similar to Fig. 3.6, the lines for GS*-Query present downward trends when μ increases from 2 to 15 on all datasets. By comparison, the changes of time cost for GS-Query and pSCAN are not obvious when increasing μ. Overall, GS*-Query significantly outperforms both GS-Query and pSCAN. Moreover, it becomes still faster when the result size decreases.

Eval-III: Query Performance on Different Datasets. The time cost for

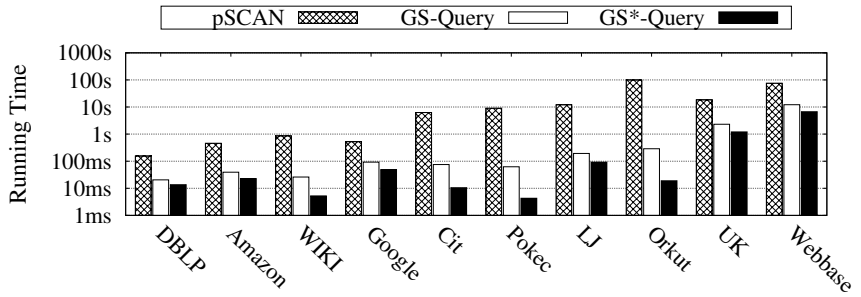


Figure 3.8: Query time on different datasets

GS*-Query, GS-Query and pSCAN under the default parameters $\mu = 5, \epsilon = 0.6$ on all datasets is reported in Fig. 3.8. We can see that GS*-Query is far more efficient than GS-Query and pSCAN on all datasets. The time cost to perform GS*-Query on *Pokec* is about only 4 milliseconds, which is the smallest value in all results. Meanwhile, GS-Query and pSCAN cost 63 milliseconds and 9 seconds respectively on that dataset. In the dataset *Webbase* with over 1 billion edges, GS*-Query spends only 6 seconds, while GS-Query and pSCAN spend 12 seconds and 76 seconds respectively.

3.6.2 Performance of Index Construction

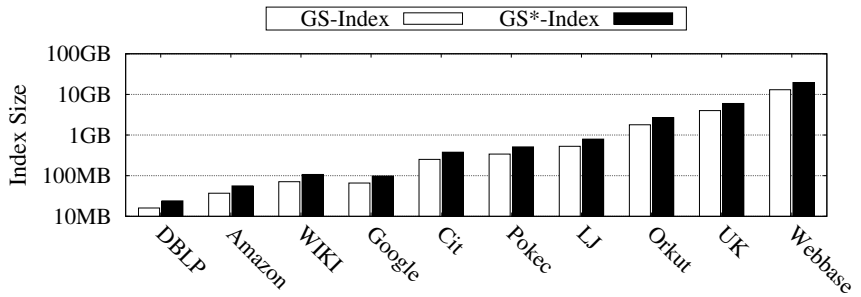


Figure 3.9: Index size for different datasets

Eval-IV: Index Size on Different Datasets. The size of GS*-Index for dif-

ferent datasets is reported in Fig. 3.9. We add GS-Index as a comparison, which only saves the structural similarity for every pair of adjacent vertices. The size of GS*-Index gradually grows when the edge number increases, and the total size of GS*-Index can be always bounded by 1.5x the size used for GS-Index. For example, in Fig. 3.9, it costs 16MB to save the GS-Index while the GS*-Index costs 24MB in DBLP. In *Webbase*, we use about 12.7GB and 19.1GB to save the GS-Index and GS*-Index respectively.

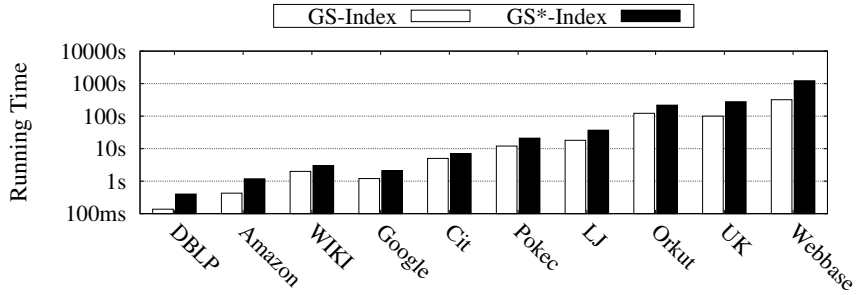
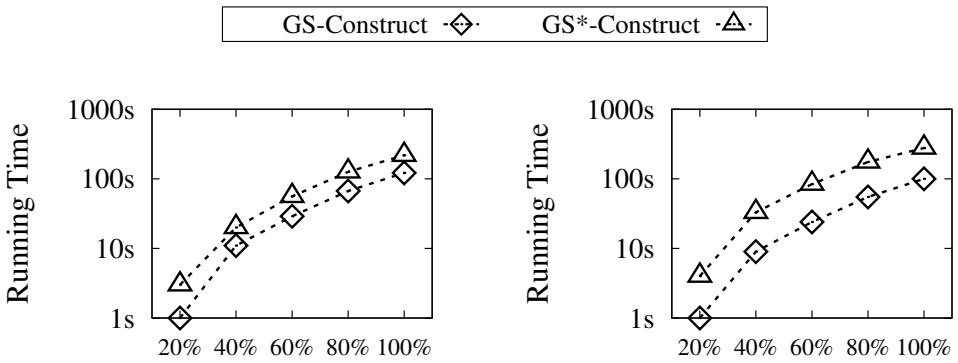


Figure 3.10: Time cost for index construction

Eval-V: Construction Time on Different Datasets. The time cost for constructing GS-Construct and GS*-Construct are both reported in Fig. 3.10. Compared with GS-Construct, we use additional time in GS*-Construct for sorting. On the largest dataset Webbase, GS*-Construct and GS-Construct cost about 20 minutes and 7 minutes respectivelly. Note that the time cost for GS*-Construct decreases from 3 seconds on WIKI to 2 seconds on Google, even though Google has more edges. This is because the number of vertices in WIKI is much larger than it is in Google.

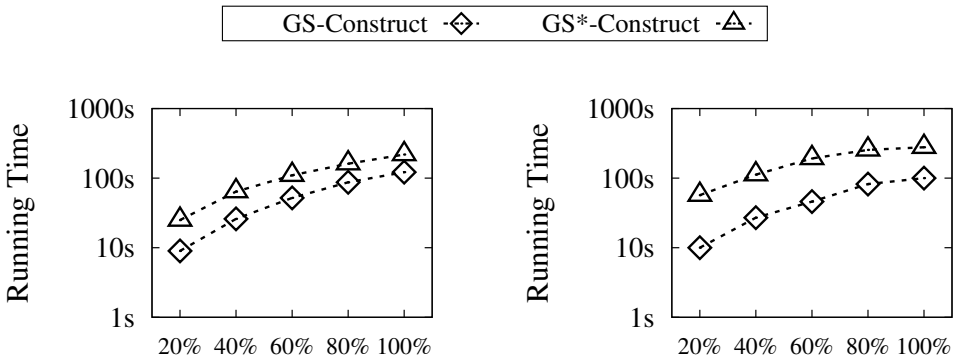
Eval-VI: Scalability Testing. We test the scalability of our proposed algorithms. We choose two real graph datasets Orkut and UK-2002 as representatives. For each dataset, we vary the graph size and graph density by randomly sampling nodes and edges respectively from 20% to 100%. When sampling nodes,

we obtain the induced subgraph of the sampled nodes, and when sampling edges, we get the incident nodes of the edges as the vertex set. We report the time cost of GS*-Construct under different percentages, with GS-Construct as a comparison. The experimental results are shown in Fig. 3.11 and Fig. 3.12.



(b) Orkut (c) UK-2002

Figure 3.11: Index construction (vary $|V|$)



(b) Orkut (c) UK-2002

Figure 3.12: Index construction (vary $|E|$)

As we can see from Fig. 3.11, GS*-Construct expends more time to sort the core-orders and neighbor-orders when the sampling ratio increases. The gap between GS*-Construct and GS-Construct remains stable. Comparing lines in

Fig. 3.11, the lines of GS*-Construct in Fig. 3.12 are gentler and have near linearly upward trends on both datasets. This demonstrates that the time cost of GS*-Construct is mainly dominated by the number of edges.

3.6.3 Performance of Index Maintenance

We test the performance of our maintenance algorithms. Since there is no previous work on this problem, we report on the algorithms GS*-Insert* and GS*-Remove*, with our basic algorithms GS*-Insert and GS*-Remove as comparisons. We randomly select 1000 distinct existing edges in the graph for each test. To test the performance of edge deletion, we remove the 1000 edges from the graphs one-by-one and record the average processing time. To test the performance of edge insertion, after the 1000 edges are removed, we insert them into the graph one by one and record the average processing time.

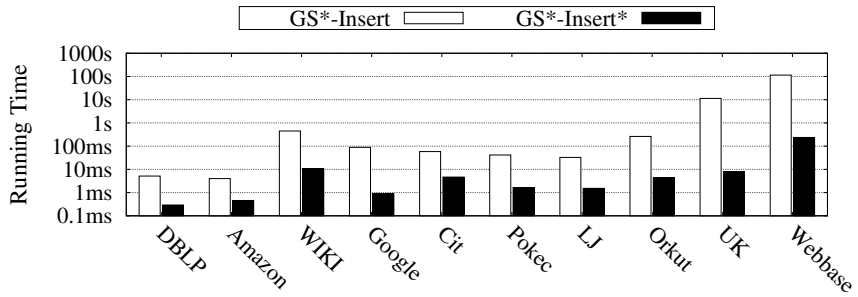


Figure 3.13: Time cost for edge insertion

Eval-VII: Index Maintenance on Different Datasets. The time cost of GS*-Insert* and GS*-Insert is reported in Fig. 3.13. We can find that GS*-Insert* is significantly faster than GS*-Insert. Of particular note, the gap increases as the graph size increases. For example, in *Webbase*, GS*-Insert spends about 114 seconds in handling the edge insertion, while GS*-Insert* only spends about 0.3 seconds. Similar results appear in Fig. 3.14 for edge removal. GS*-Remove* is

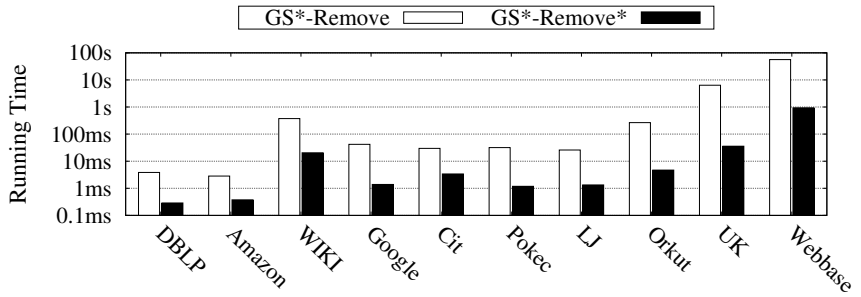
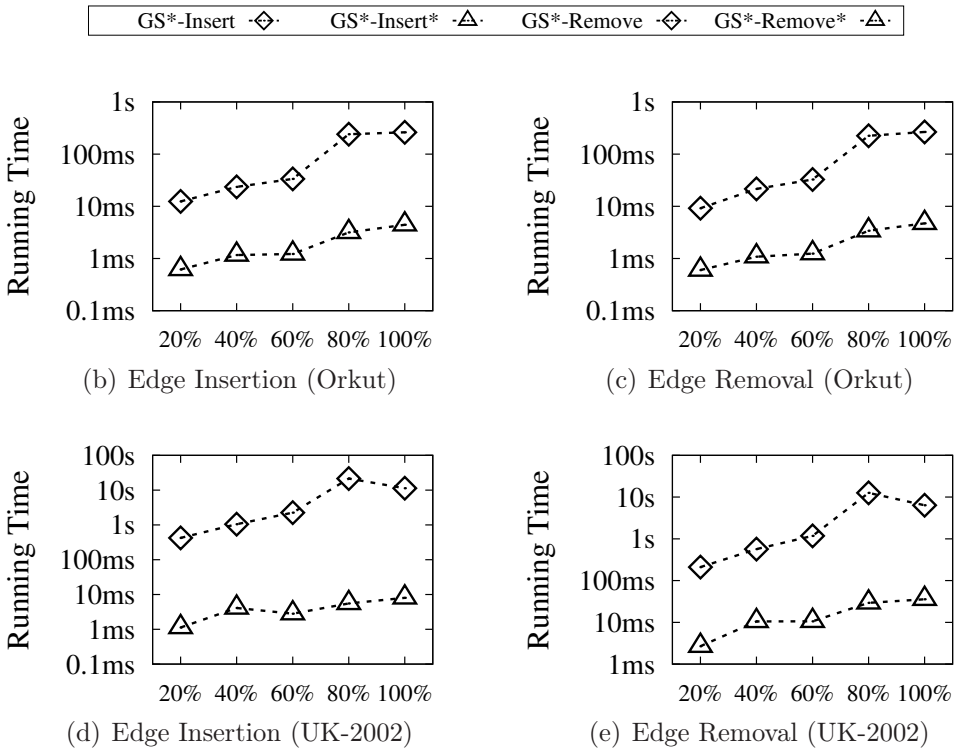


Figure 3.14: Time cost for edge removal

at least about 10x faster than GS*-Remove on all datasets and in some specific graphs, such as UK-2002, GS*-Remove spends more than 6 seconds compared to GS*-Remove*'s 35 milliseconds. The result shows our proposed algorithms GS*-Insert* and GS*-Remove* are significantly more efficient than GS*-Insert and GS*-Remove.

Eval-VIII: Scalability Testing. We evaluate the index-maintenance algorithm's performance by sampling $|V|$ and sampling $|E|$ respectively. The sampling method is the same as that in Eval-VI of Subsection 3.6.2. The statistics for varying $|V|$ and varying $|E|$ are reported in Fig. 3.15 and Fig. 3.16 respectively. All figures show that the time cost of GS*-Insert* (resp. GS*-Remove*) is significantly less than that of GS*-Insert (resp. GS*-Remove) and the gap between them grows when the sampling ratio increases. The increase of GS*-Insert* and GS*-Remove* is not obvious when graph size increases especially in Fig. 3.16. This suggests that the time cost for maintenance algorithms does not closely rely on the graph size. Overall, the results imply that our optimization techniques are effective and our final algorithms are efficient.



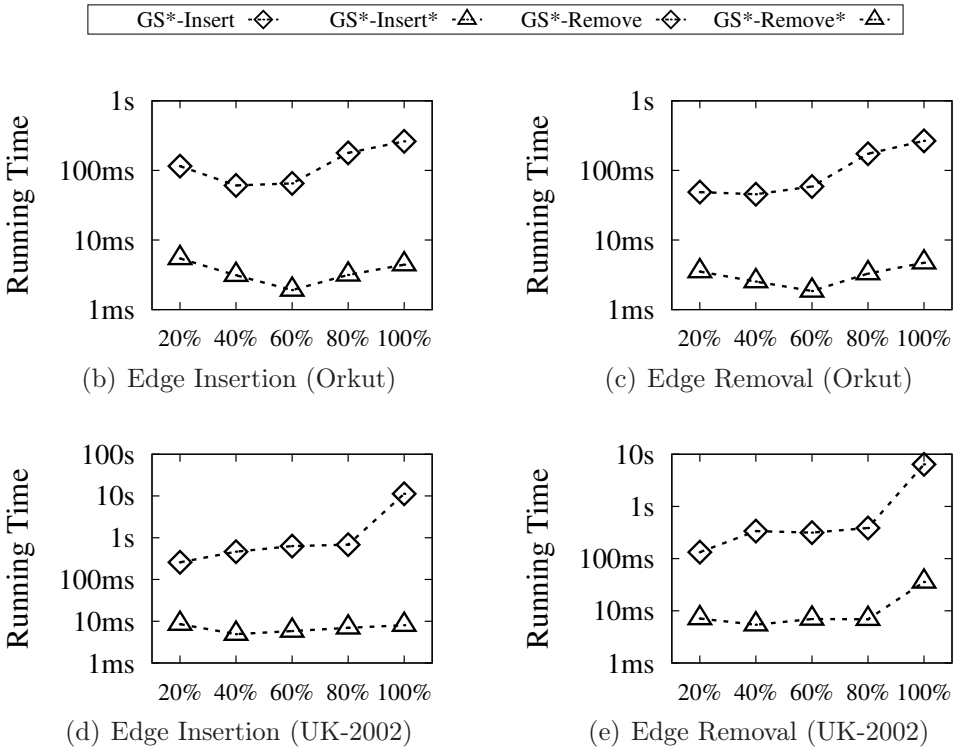
(b) Edge Insertion (Orkut) (c) Edge Removal (Orkut)

(d) Edge Insertion (UK-2002) (e) Edge Removal (UK-2002)

Figure 3.15: Index maintenance (vary $|V|$)

3.7 Chapter Summary

In this chapter, we introduce an index-based method for structural graph clustering. Our proposed index, named GS*-Index, contains two parts, which are core-orders and neighbor orders. We use core-orders to efficiently detect all cores. We use neighbor-orders to group all cores and construct the clusters. Based on GS*-Index, we efficiently answer the query for any given ϵ and μ, and the time cost for the query algorithm is only proportional to the size of clusters. To handle graph updates, we propose index maintenance algorithms. The experimental results demonstrate that our solution significantly outperforms the state-of-the-art algorithm.



(b) Edge Insertion (Orkut) (c) Edge Removal (Orkut)

(d) Edge Insertion (UK-2002) (e) Edge Removal (UK-2002)

Figure 3.16: Index maintenance (vary $|E|$)

Chapter 4

DEGREE-CONSTRAINED COMMUNITY DETECTION

4.1 Chapter Overview

In this chapter, we introduce the I/O efficient algorithm for the core graph decomposition. Note that the term core has been defined to represent a special type of vertices in SCAN model in Chapter 3, while the core used in this chapter only represents the concept of k-core. The work is published in [86] and the rest of this chapter is organized as follows. Section 4.2 gives preliminary definitions and formally defines the problem. Section 4.3 reviews existing solutions. Section 4.4 introduces our semi-external core decomposition algorithm. Section 4.5 describes the I/O efficient core maintenance algorithms. Section 4.6 presents the extension for graph degeneracy ordering. Section 4.7 practically evaluates our proposed algorithms. Section 4.8 summarizes the chapter.

4.2 Preliminary

Given a graph G, the formal definition of the k-core is given as follows.

Definition 11. ***(k-Core)*** *Given a graph G and an integer k, the k-core of graph G, denoted by G_k, is a maximal subgraph of G in which every vertex has a degree of at least k, i.e., $\forall v \in V(G_k), d(v, G_k) \geq k$ [69].*

Let k_{max} be the maximum possible k value such that a k-core of G exists. According to [9], the k-cores of graph G for all $1 \leq k \leq k_{max}$ have the following property:

Property 1. $\forall 1 \leq k < k_{max} : G_{k+1} \subseteq G_k$.

Next, we define the core number for each $v \in V(G)$.

Definition 12. ***(Core Number)*** *Given a graph G, for each vertex $v \in V(G)$, the core number of v, denoted by $core(v, G)$, is the largest k, such that v is contained in a k-core, i.e., $core(v, G) = \max\{k | v \in V(G_k)\}$. For simplicity, we use $core(v)$ to denote $core(v, G)$ if the context is self-evident.*

Based on Property 1 and Definiton 12, we can easily derive the following lemma:

Lemma 8. *Given a graph G and an integer k, let $V_k = \{v \in V(G) | core(v) \geq k\}$, we have $G_k = G(V_k)$.*

Problem Statement. We study the problem of core decomposition, which is defined as follows: Given a graph G, core decomposition computes the k-cores of G for all $1 \leq k \leq k_{max}$. We also consider how to update the k-cores of G for all $1 \leq k \leq k_{max}$ incrementally when G is dynamically updated by insertion and deletion of edges.

According to Lemma 8, core decomposition is equivalent to computing $core(v)$ for all $v \in V(G)$. Therefore in this chapter, we study how to compute $core(v)$ for all $v \in V(G)$ and how to maintain them incrementally when graph dynamically updates .

Considering that many real-world graphs are huge and cannot entirely reside in main memory, we aim to design I/O efficient algorithms to compute and maintain the core numbers of all vertices in the graph G. To analyze the algorithm, we use the external memory model introduced in [1]. Let M be the size of main memory and let B be the disk block size. A read I/O will load one block of size B from disk into main memory, and a write I/O will write one block of size B from the main memory into disk.

Assumption. In this chapter, we follow a semi-external model by assuming that the vertices can be loaded in main memory while the edges are stored on disk, i.e., we assume that $M \geq c \times |V(G)|$ where c is a small constant. This assumption is practical because in most social networks and web graphs, the number of edges is much larger than the number of vertices.

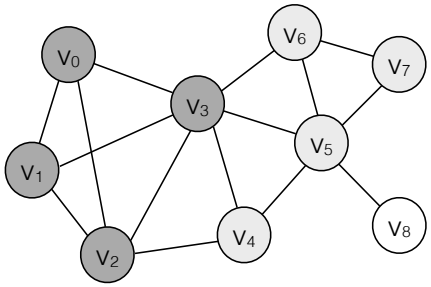


Figure 4.1: A sample graph G and its core decomposition

Example 5. *Consider the graph G in Fig. 4.1, the induced subgraph of* $\{v_0, v_1, v_2, v_3\}$ *is a* 3*-core in which every vertex has a degree at least* 3*. Since no* 4*-core exists, we have* $core(v_0) = core(v_1) = core(v_2) = core(v_3) = 3$*. Similarly, we*

can derive that $core(v_4) = core(v_5) = core(v_6) = core(v_7) = 2$ *and* $core(v_8) = 1$. *When an edge* (v_7, v_8) *is inserted into* G, $core(v_8)$ *increases from* 1 *to* 2, *and the core numbers of other vertices keep unchanged.*

4.3 Existing Solutions

In this section, we introduce three state-of-the-art solutions for in-memory core decomposition, I/O efficient core decomposition and in-memory core maintenance respectively.

In-memory Core Decomposition. The state-of-the-art in-memory core decomposition algorithm, denote by IMCore, is proposed in [9]. The pseudocode of IMCore is shown in Algorithm 9. The algorithm processes the vertex with core number k in increasing order of k. Each time, k is selected as the minimum degree of current vertices in the graph (line 3). Whenever there exists a vertex v with degree no larger than k in the graph (line 4), we can guarantee that the core number of v is k (line 5) and we remove v with all its incident edges from the graph (line 6). Finally, the core numbers of all vertices are returned (line 7). With the help of bin sort to maintain the minimum degree of the graph, IMCore can achieve a time complexity of $O(m + n)$, which is optimal.

Algorithm 9 IMCore(Graph G)

```
1: G' ← G;
2: while G' ≠ ∅ do
3:     k ← min_{v∈V(G')} deg(v, G');
4:     while ∃v ∈ V(G') : deg(v, G') ≤ k do
5:         core(v) ← k;
6:         remove v and its incident edges from G';
7: return core(v) for all v ∈ V(G);
```

I/O Efficient Core Decomposition. The state-of-the-art efficient core decomposition algorithm is proposed in [19]. The algorithm, denoted as EMCore,

is shown in Algorithm 10. It first divides the whole graph G into partitions on disk (line 1). Each partition contains a disjoint set of vertices along with their incident edges. An upper bound of $core(v)$, denoted by $ub(v)$, is computed for each vertex v in each partition P_i. Then the algorithm iteratively computes the core numbers for vertices in a top-down manner.

Algorithm 10 EMCore(Graph G on Disk)

```
1: divide G into partitions P = {P_1, P_2, ..., P_t} on disk;
2: for each partition P_i ∈ P do
3:     compute ub(v) for all v ∈ V(P_i);
4: k_u ← +∞;
5: while k_u > 0 do
6:     estimate k_l;
7:     P_mem ← {P_i ∈ P | ∃v ∈ V(P_i) : ub(v) ∈ [k_l, k_u]};
8:     G_mem ← load partitions in P_mem in main memory;
9:     core(v) ← core(v, G_mem) for all core(v, G_mem) ∈ [k_l, k_u];
10:    for each partition P_i ∈ P_mem do
11:        remove vertices v with core(v, G_mem) ∈ [k_l, k_u] from P_i;
12:        update ub(v) and deg(v) for all v ∈ V(P_i);
13:        write P_i back to disk (merge small partitions if necessary);
14:    k_u ← k_l − 1;
15: return core(v) for all v ∈ V(G);
```

In iteration, the vertices with core values falling in a certain range $[k_l, k_u]$ are computed (line 6-14). Here, k_l is estimated based on the number of partitions that can be loaded in main memory (line 6). In line 7, the algorithm computes the set of partitions each of which contains at least one vertex v with $ub(v)$ falling in $[k_l, k_u]$, and in line 8, all such partitions are loaded in main memory to form an in-memory graph G_{mem}. In line 9, an in-memory core decomposition algorithm is applied on G_{mem}, and those vertices in G_{mem} with core numbers falling in $[k_l, k_u]$ get their exact core numbers in G. After that, for all partitions loaded in memory (line 10), those vertices with exact core numbers computed are removed from the partition (line 11), and their core number upper bounds and degrees are updated accordingly (line 12). Here the new vertex degrees have to consider the deposited degrees from the removed vertices. Finally, the

in-memory partitions are merged and written back to disk (line 13), and k_u is set to be $k_l - 1$ to process the next range of k values in the next iteration.

The I/O complexity of EMCore is $O(\frac{d \cdot (m+n)}{B})$. The CPU complexity of EMCore is $O(d \cdot (m + n))$. However, the space complexity of EMCore cannot be well bounded. In the worst case, it still requires $O(m + n)$ memory space to load the whole graph into main memory. Therefore, EMCore is not scalable to handle large-sized graphs.

In-memory Core Maintenance. To handle the case when the graph is dynamically updated by insertion and deletion of edges, the state-of-the-art core maintenance algorithms are proposed in [68] and [47], which are based on the same findings shown in the following theorems:

Theorem 7. *If an edge is inserted into (deleted from) graph G, the core number $core(v)$ for any $v \in V(G)$ may increase (decrease) by at most* 1.

Theorem 8. *If an edge (u, v) is inserted into (deleted from) graph G, suppose $core(v) \leq core(u)$ and let V' be the set of vertices whose core numbers have changed, if $V' \neq \emptyset$, we have:*

- *$G(V')$ is a connected subgraph of G;*
- *$v \in V'$; and*
- *$\forall v' \in V' : core(v') = core(v)$;*

Based on Theorem 7 and Theorem 8, after an edge (u, v) is inserted into (deleted from) graph G, suppose $core(v) \leq core(u)$, instead of computing the core numbers for all vertices in G from scratch, we can restrain the core computation within a small range of vertices V' in G. Specifically, we can follow a two-step approach: In the first step, we can perform a depth-first-search from vertex v in G to compute all vertices v' with $core(v') = core(v)$ that are reachable from v via a path that consists of vertices with core numbers equal to $core(v)$.

Such vertices form a set V' which is usually much smaller than $V(G)$. In the second step, we only restrain the core number updates within the subgraph $G(V')$ in memory, and each update increases (decreases) the core number of a vertex by at most 1. The algorithm details and other optimization techniques can be found in [68] and [47].

4.4 I/O Efficient Core Decomposition

4.4.1 Basic Semi-external Algorithm

Drawback of EMCore. EMCore (Algorithm 10) is the state-of-the-art I/O efficient core decomposition algorithm. However, EMCore cannot be used to handle big graphs, since the number of partitions to be loaded into main memory in each iteration cannot be well-bounded. In line 7-8 of Algorithm 10, as long as a partition contains a vertex v with $ub(v) \in [k_l, k_u]$, the whole partition needs to be loaded into main memory. When k_u becomes small, it is highly possible for a partition to contain a vertex v with $ub(v) \in [k_l, k_u]$. Consequently, almost all partitions are loaded into main memory. Due to this reason, the space used for EMCore is $O(m+n)$, and it cannot be significantly reduced in practice.

Locality Property. We aim to design a semi-external algorithm for core decomposition. Note that the graph is stored in disk by adjacency list. First, we introduce a locality property [60] for core numbers below:

Theorem 9. ***(Locality)*** *Given a vertex v, the core number $core(v) = k$ if and only if:*

• *There exists a subset $V_k \subseteq N(v)$ such that $|V_k| = core(v)$ and $\forall u \in V_k$: $core(u) \geq core(v)$; and*

• *There does not exist a subset* $V_{k+1} \subseteq N(v)$ *such that* $|V_{k+1}| = core(v) + 1$ *and* $\forall u \in V_{k+1} : core(u) \geq core(v) + 1$.

Based on Theorem 9, $core(v)$ can be calculated using the following recursive equation:

$$core(v) = \max k \text{ s.t. } |\{u \in N(v) | core(u) \geq k\}| \geq k \tag{4.1}$$

Based on Theorem 9, a distributed algorithm is designed in [60], in which each vertex v initially assigns its core number as an arbitrary core number upper bound (e.g., $\deg(v)$), and keeps updating its core numbers using Eq. 4.1 until convergence.

Algorithm 11 SemiCore(graph G on disk)

```
1: core(v) ← deg(v) for all v ∈ V(G);
2: update ← true;
3: while update do
4:     update ← false;
5:     for v ← v_1 to v_n do
6:         load N(v) from disk;
7:         c_old ← core(v);
8:         core(v) ← LocalCore(c_old, N(v));
9:         if core(v) ≠ c_old then update ← true;
10: return core(v) for all v ∈ V(G);
11: Procedure LocalCore(c_old, N(v))
12: num(i) ← 0 for all 1 ≤ i ≤ c_old;
13: for each u ∈ N(v) do
14:     i ← min{c_old, core(u)};
15:     num(i) ← num(i) + 1;
16: s ← 0;
17: for k ← c_old to 1 do
18:     s ← s + num(k);
19:     if s ≥ k then break;
20: return k;
```

Basic Solution. We make use of the locality property to design a semi-external algorithm for core decomposition. The algorithm is shown in Algorithm 11. Here, we use $\overline{\mathsf{core}}(v)$ to denote the intermediate core number for v, which is

always an upper bound of $core(v)$ and will finally converge to $core(v)$. Initially, $\overline{\mathsf{core}}(v)$ is assigned as an arbitrary upper bound of $core(v)$ (e.g., $\deg(v)$). Then, we iteratively update $\overline{\mathsf{core}}(v)$ for all $v \in V(G)$ using the locality property until convergence (lines 2-9).

In each iteration (lines 5-9), we sequentially scan the vertex table on disk to get the offset and $deg(v)$ for each vertex v from v_1 to v_n (line 5). Then we load $N(v)$ from disk using the offset and $deg(v)$ for each such vertex v, (line 6). Recall that the edge table on disk stores $N(v)$ from v_1 to v_n sequentially. Therefore, we can load $N(v)$ easily using sequential scan of the edge table on disk. In line 7-9, we record the original core number c_{old} of v (line 7); compute an updated core number of v using Eq. 4.1 by invoking $\mathsf{LocalCore}(c_{old}, N(v))$ (line 8); and continue the iteration if $\overline{\mathsf{core}}(v)$ is updated (line 9). Finally, when $\overline{\mathsf{core}}(v)$ for all $v \in V(G)$ keeps unchanged, we return them as their core numbers (line 10).

The procedure $\mathsf{LocalCore}(c_{old}, N(v))$ to compute the new core number of v using Eq. 4.1 is shown in lines 11-20 of Algorithm 11. We use $\mathsf{num}(i)$ to denote the number of neighbors of v with $\overline{\mathsf{core}}$ equals i (if $i < c_{old}$) or no smaller than i (if $i = c_{old}$) (lines 12-15). After computing $\mathsf{num}(i)$ for all $1 \leq i \leq c_{old}$, we decrease k from c_{old} to 1 (line 17), and for each k, we compute the number of neighbors of v with $\overline{\mathsf{core}} \geq k$, denoted by s (line 18), i.e., $s = |\{u \in N(v) | \overline{\mathsf{core}}(u) \geq k\}|$. Once $s \geq k$, we get the maximum k with $|\{u \in N(v) | \overline{\mathsf{core}}(u) \geq k\}| \geq k$, and we return k as the new core number (line 20). Since $c_{old} \leq N(v)$, the time complexity of $\mathsf{LocalCore}(c_{old}, N(v))$ is $O(deg(v))$.

Algorithm Analysis. Let l be the number of iterations of Algorithm 11. The space, CPU time, I/O complexity of Algorithm 11 are shown below (B is the block size):

Theorem 10. *The space, I/O, and time comlexities of Algorithm 11 are* $O(n)$, $O(\frac{l \cdot (m+n)}{B})$, *and* $O(l \cdot (m+n))$ *respectively.*

Discussion. Note that we use a value l to denote the number of iterations of Algorithm 11. Although l is bounded by n as proved in [60], it is much smaller in practice and is usually not largely influenced by the size of the graph. For example, in a social network Twitter with $n = 41.7$ M, $m = 1.47$ G, and $k_{max} = 2488$ used in our experiments, the number of iterations using Algorithm 11 is only 62. In a web graph UK-2002 with $n = 105.9$ M, $m = 3.74$ G, and $k_{max} = 5704$ used in our experiments, the number of iterations is 2137. In the largest dataset Clueweb with $n = 978.4$ M, $m = 42.57$ G, and $k_{max} = 4244$ used in our experiments, the number of iterations is only 943.

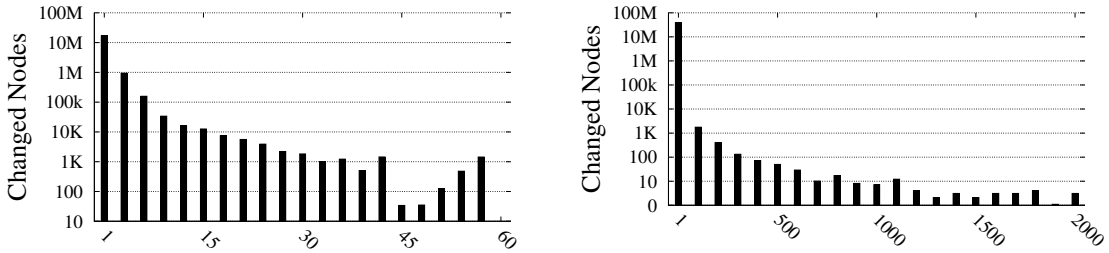
Iteration\v	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
Init	3	3	4	6	3	5	3	2	1
Iteration 1	3	3	3	3	3	3	2	2	1
Iteration 2	3	3	3	3	3	2	2	2	1
Iteration 3	3	3	3	3	2	2	2	2	1
Iteration 4	3	3	3	3	2	2	2	2	1

Table 4.1: Illustration of SemiCore

Example 6. *The process to compute the core numbers for vertices in Fig. 4.1 using Algorithm 11 is shown in Table 4.1. The number in each cell is the value* $\overline{\mathsf{core}}(v_i)$ *for the corresponding vertex* v_i *in each iteration. The grey cells are those whose upper bounds is computed through invoking* LocalCore*. In iteration 1, when processing* v_3*, the* $\overline{\mathsf{core}}$ *values for the neighbors of* v_3 *are* $\{3, 3, 3, 3, 5, 3\}$*. There are* 3 *neighbors with* $\overline{\mathsf{core}} \geq 3$ *but no* 4 *neighbors with* $\overline{\mathsf{core}} \geq 4$*. Therefore,* $\overline{\mathsf{core}}(v_3)$ *is updated from* 6 *to* 3*. The algorithm terminates in* 4 *iterations.*

4.4.2 Optimal Vertex Computation

In previous subsection, for all vertices $v \in V(G)$ in each iteration of Algorithm 11, the neighbors of v are loaded from disk and $\overline{\mathsf{core}}(v)$ is recomputed. However, if we can guarantee that $\overline{\mathsf{core}}(v)$ is unchanged after recomputation, there is no need to recompute $\overline{\mathsf{core}}(v)$ by invoking LocalCore.



(a) Twitter (Vary Iteration) (b) UK-2002 (Vary Iteration)

Figure 4.2: Number of vertices whose core numbers are changed

To illustrate the effectiveness of eliminating such useless vertex computation, in Fig. 4.2, we show the number of vertices whose $\overline{\mathsf{core}}$ values are updated in each iteration for the Twitter and UK-2002 datasets used in our experiments. For the Twitter dataset, the algorithm runs for a total of 62 iterations. In iteration 1, 10 M vertices have their core numbers updated. However, in iteration 5, only 1 M vertices have their core numbers updated, which is only 10% of the number in iteration 1. From iteration 30 on, less then 2 K vertices have their core numbers updated in each iteration. For the UK-2002 dataset, the observation is similar. A total of 2137 iterations are executed. The number of core number updates in iteration 1 is 10^4 times larger than that in iteration 100, and from iteration 400 to iteration 2137, less than 100 vertices have their core numbers updated in each iteration.

The above observations indicate that reducing the number of useless vertex computations can largely improve the performance of the algorithm. In this

section, we aim to design an optimal vertex computation scheme that guarantees that every execution of LocalCore will update the core number.

The Rationale. Our general idea is to maintain more vertex information which can be used to check whether a vertex computation is needed. Note that $\overline{\mathsf{core}}(v)$ for each $v \in V(G)$ will never increase, and according to Eq. 4.1, $core(v)$ is determined by the number of neighbors u with $core(u) \geq core(v)$. Therefore, for each vertex v in the graph, we maintain the number of such neighbors, denoted by $cnt(v)$, which is defined as follows:

$$cnt(v) = |\{u \in N(v) | \overline{\mathsf{core}}(u) \geq \overline{\mathsf{core}}(v)\}| \tag{4.2}$$

With the assistance of $cnt(v)$ for all $v \in V(G)$, we can derive a sufficient and necessary condition for the core number of a vertex to be updated using the following lemma:

Lemma 9. *For each vertex $v \in V(G)$, $\overline{\mathsf{core}}(v)$ is updated if and only if $cnt(v) < \overline{\mathsf{core}}(v)$.*

Proof. We first prove $\Leftarrow$: Suppose $cnt(v) < \overline{\mathsf{core}}(v)$, we have $|\{u \in N(v) | \overline{\mathsf{core}}(u) \geq \overline{\mathsf{core}}(v)\}| < \overline{\mathsf{core}}(v)$. Consequently, $\overline{\mathsf{core}}(v)$ needs to be decreased by at least 1 to satisfy Eq. 4.1.

Next, we prove $\Rightarrow$: Suppose $\overline{\mathsf{core}}(v)$ needs to be updated. According to Eq. 4.1, either $|\{u \in N(v) | \overline{\mathsf{core}}(u) \geq \overline{\mathsf{core}}(v)\}| < \overline{\mathsf{core}}(v)$ or there is a larger k s.t. $|\{u \in N(v) | \overline{\mathsf{core}}(u) \geq k\}| \geq k$. The latter is impossible since $\overline{\mathsf{core}}(u)$ will never increase during the algorithm. Therefore, $cnt(v) < \overline{\mathsf{core}}(v)$. □

Algorithm Design. Based on the above discussion, we propose a new algorithm SemiCore* with optimal vertex computation. The algorithm is shown in Algorithm 12. In line 1-3, we initialize $\overline{\mathsf{core}}(v)$, active$(v)$ and update. For $cnt(v)$ ($v \in V(G)$), we initialize it to be 0 which will be updated to its real value

after the first iteration. In each iteration, we partially scan the graph on disk (line 6-11). Specifically, for each vertex v, the condition to load $N(v)$ from disk is $cnt(v) < \overline{\mathsf{core}}(v)$ according to Lemma 9 (line 6-8). In line 9, we compute the new $\overline{\mathsf{core}}(v)$, and we can guarantee that $\overline{\mathsf{core}}(v)$ will decrease by at least 1. In line 10, we compute $cnt(v)$ by invoking $\mathsf{ComputeCnt}(N(v), \overline{\mathsf{core}}(v))$ (line 13-17) which follows Eq. 4.2. In line 11, since $\overline{\mathsf{core}}(v)$ has been decreased from c_{old}, we need to update $cnt(u)$ for every $u \in N(v)$ by invoking $\mathsf{UpdateNbrCnt}(N(v), c_{old}, \overline{\mathsf{core}}(v))$ (line 18-20). Here, according to Eq. 4.2, only those vertices u with $\overline{\mathsf{core}}(u)$ falling in the range $(\overline{\mathsf{core}}(v), c_{old}]$ will have $cnt(u)$ decreased by 1 (line 20). Finally, after the algorithm converges, the final core numbers for vertices in the graph are returned (line 12).

Algorithm 12 $\mathsf{SemiCore}^*$(graph G on disk)

```
1: core(v) ← deg(v) for all v ∈ V(G);
2: cnt(v) ← 0 for all v ∈ V(G);
3: update ← true;
4: while update do
5:     update ← false;
6:     for v ← v_1 to v_n s.t. cnt(v) < core(v) do
7:         update ← true;
8:         load N(v) from disk;
9:         c_old ← core(v); core(v) ← LocalCore(c_old, N(v));
10:        cnt(v) ← ComputeCnt(N(v), core(v));
11:        UpdateNbrCnt(N(v), c_old, core(v));
12: return core(v) for all v ∈ V(G);
13: Procedure ComputeCnt(N(v), core(v))
14: s ← 0;
15: for each u ∈ N(v) do
16:     if core(u) ≥ core(v) then s ← s + 1;
17: return s;
18: Procedure UpdateNbrCnt(N(v), c_old, core(v))
19: for each u ∈ N(v) do
20:     if core(v) < core(u) ≤ c_old then cnt(u) ← cnt(u) − 1;
```

Compared to Algorithm 11, Algorithm 12 can largely reduce the number of vertex computations since Algorithm 12 only computes the core number of a vertex whenever necessary. On the other hand, for each vertex v to be computed, in

addition to invoking LocalCore, Algorithm 12 takes extra cost to maintain $cnt(v)$ using ComputeCnt, and update $cnt(u)$ for $u \in N(v)$ using UpdateNbrCnt. However, it is easy to see that both ComputeCnt and UpdateNbrCnt take $O(deg(v))$ time which is the same as the time complexity of LocalCore. Therefore, the extra cost can be well bounded.

Algorithm Analysis. Compared to the state-of-the-art I/O efficient core decomposition algorithm EMCore, SemiCore* (Algorithm 12) has the following advantages:

A_1*: Bounded Memory.* SemiCore* follows the semi-external model and requires only $O(n)$ memory while EMCore requires $O(m+n)$ memory in the worst case. For instance, to handle the Orkut dataset with 3 M vertices and 117.2 M edges used in our experiments, SemiCore* consumes 12 M memory; EMCore consumes 938 M memory; and the in-memory algorithm IMCore (Algorithm 9) consumes 1070 M memory.

A_2*: Read I/O Only.* In SemiCore*, we only require read I/Os by scanning the vertex and edge tables sequentially on disk in each iteration. However, EMCore needs both read and write I/Os since the partitions loaded into main memory will be repartitioned and written back to disk in each iteration. In practice, a write I/O is usually much slower than a read I/O.

A_3*: Simple In-memory Structure and Data Access.* In EMCore, it invokes the in-memory algorithm IMCore that uses a complex data structure for bin sort. It also involves complex graph partitioning and repartitioning algorithms. In SemiCore*, we only use two arrays *core* and *cnt*, and the data access is simple. This makes SemiCore* very efficient in practice and even more efficient than the in-memory algorithm IMCore in many datasets. For instance, to handle the Orkut dataset used in our experiments, EMCore, IMCore, and SemiCore* consumes 63.2 seconds, 18.4 seconds, and 16.3 seconds respectively.

Iteration\v	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
Init	3	3	4	6	3	5	3	2	1
Iteration 1	3	3	3	3	3	3	2	2	1
Iteration 2	3	3	3	3	3	2	2	2	1
Iteration 3	3	3	3	3	2	2	2	2	1

Table 4.2: Illustration of SemiCore*

Example 7. *The process to handle the graph G in Fig. 4.1 using Algorithm 12 is shown in Table 4.2. We show $\overline{\mathsf{core}}(v)$ each $v \in V(G)$ in each iteration, and those recomputed $\overline{\mathsf{core}}(v)$ values are shown in grey cells. For instance, after iteration 1, we have $\overline{\mathsf{core}}(v_5) = 3$ and $cnt(v_5) = 2$ since only its two neighbors v_3 and v_4 have their $\overline{\mathsf{core}}$ values no smaller than 3. Therefore, in iteration 2, $\overline{\mathsf{core}}(v_5)$ is recomputed and updated from 3 to 2. This also updates the cnt value of its neighbor v_4 from 3 to 2 since $\overline{\mathsf{core}}(v_4) = 3$. Note that in iteration 1, we need to compute $\overline{\mathsf{core}}(v)$ for all $v \in V(G)$ since $cnt(v)$ is unknown only in the first iteration. Compared to Algorithm 11 in Example 6, Algorithm 12 only uses 3 iterations and reduces the number of vertex computations from 23 to 11.*

4.5 I/O Efficient Core Maintenance

In this section, we discuss how to incrementally maintain the core numbers when edges are inserted into or deleted from the graph under the semi-external setting.

4.5.1 Edge Deletion

Algorithm Design. From Theorem 7, we know that after an edge deletion, the core number for any $v \in V(G)$ will decrease by at most 1. Therefore, after an edge deletion, the old core numbers of vertices in the graph are upper bounds

of their new core numbers. Recall that in Algorithm 12, as long as $\overline{\mathsf{core}}(v)$ is initialized to be an arbitrary upper bound of $core(v)$ for all $v \in V(G)$, $\overline{\mathsf{core}}(v)$ can be converged to $core(v)$ after the algorithm terminates. Therefore, Algorithm 12 can be easily modified to handle edge deletion.

Algorithm 13 SemiDelete*(graph G on disk, edge (u, v))

```
1: delete (u, v) from G;
2: if core̅(u) < core̅(v) then
3:     cnt(u) ← cnt(u) − 1;
4: else if core̅(v) < core̅(u) then
5:     cnt(v) ← cnt(v) − 1;
6: else
7:     cnt(u) ← cnt(u) − 1; cnt(v) ← cnt(v) − 1;
8: line 3-12 of Algorithm 12;
```

Specifically, we show our algorithm SemiDelete* for edge deletion in Algorithm 13. Given an edge $(u, v) \in E(G)$ to be removed, we first delete (u, v) from G (line 1). In line 2-7, we update $cnt(u)$ and $cnt(v)$ due to the deletion of edge (u, v). Here, we consider three cases. First, if $\overline{\mathsf{core}}(u) < \overline{\mathsf{core}}(v)$, we only need to decrease $cnt(u)$ by 1. Second, if $\overline{\mathsf{core}}(v) < \overline{\mathsf{core}}(u)$, we decrease $cnt(v)$ by 1. Third, if $\overline{\mathsf{core}}(v) = \overline{\mathsf{core}}(u)$, we decrease both $cnt(v)$ and $cnt(u)$ by 1. Now we can use Algorithm 12 to update the core numbers of other vertices (line 8).

Iteration\v	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
Old Value	3	3	3	3	2	2	2	2	1
Iteration 1	2	2	2	2	2	2	2	2	1

Table 4.3: Illustration of SemiDelete* (delete (v_0, v_1))

Example 8. *Suppose after Example 7, we delete edge (v_0, v_1) from G (Fig. 4.1). By Algorithm 13, we update $cnt(v_0)$ and $cnt(v_1)$ from 3 to 2 and invoke line 3-12 of Algorithm 12. Only 1 iteration is needed with 4 vertex computations as shown in Table 4.3.*

Algorithm Analysis. Algorithm 13 performs the same operations as Algorithm 12. Therefore, it can be obviously bounded by the same space, CPU time and I/O complexities as in Theorem 10. For a removed edge (u, v), without loss of generality, we assume $core(u) \leq core(v)$. Let l be the number of iterations of Algorithm 13 and V_c be the set of vertices reachable from u via a path that consists of vertices w with $core(w) = core(u)$, we have:

Theorem 11. *The space, I/O, and time complexities of Algorithm 13 are* $O(n)$, $O(\frac{l \cdot (m+n)}{B})$, *and* $O(\sum_{w \in V_c} deg(w) + l \cdot n)$ *respectively.*

The iteration number l for Algorithm 13 is bounded by $|V_c|$, which is much smaller than n in practice.

4.5.2 Edge Insertion

The Rationale. After a new edge (u, v) is inserted into graph G, according to Theorem 7, we know that the core number for any $v \in V(G)$ will increase by at most 1. As a result, the old core number of a vertex in the graph may not be an upper bound of its new core number. Therefore, Algorithm 12 cannot be applied directly to handle edge insertion. However, according to Theorem 8, after inserting an edge (u, v) (suppose $\overline{\mathsf{core}}(v) \leq \overline{\mathsf{core}}(u)$), we can find a candidate set V_c consisting of all vertices w that are reachable from vertex v via a path that consists of vertices with $\overline{\mathsf{core}}$ equals $\overline{\mathsf{core}}(v)$, and we can guarantee that those vertices with core numbers increased by 1 is a subset of V_c. Consequently, if we increase $\overline{\mathsf{core}}(v)$ by 1 for all $v \in V_c$, we can guarantee that for all $u \in V(G)$, $\overline{\mathsf{core}}(u)$ is an upper bound of the new core number of u. Thus we can apply Algorithm 12 to compute the new core numbers.

Algorithm Design. Our algorithm SemiInsert for edge insertion is shown in Algorithm 14. In line 1, we insert (u, v) into G. In line 1-4, we update $cnt(u)$

Algorithm 14 SemiInsert(graph G on disk, edge (u, v))

```
1: insert (u, v) into G;
2: swap u and v if core(u) > core(v);
3: cnt(u) ← cnt(u) + 1;
4: if core(v) = core(u) then cnt(v) ← cnt(v) + 1;
5: c_old ← core(u);
6: active(w) ← false for all w ∈ V(G); active(u) ← true;
7: update ← true;
8: while update do
9:     update ← false;
10:    for v' ← v_1 to v_n do
11:        if active(v') = true and core(v') = c_old then
12:            update ← true
13:            core(v') ← core(v') + 1;
14:            load N(v') from disk;
15:            cnt(v') ← ComputeCnt(N(v'), core(v'));
16:            for each u' ∈ N(v') s.t. core(u') = core(v') do
17:                cnt(u') ← cnt(u') + 1;
18:            for each u' ∈ N(v') do
19:                if core(u') = c_old and active(u') = false then
20:                    active(u') ← true;
21: line 3-12 of Algorithm 12;
```

and $cnt(v)$ caused by the insertion of edge (u, v). We use $\mathsf{active}(w)$ to denote whether w is a candidate vertex with core number increased which is initialized to be false except for vertex u. In line 8-20, we iteratively update $\mathsf{active}(w)$ for $w \in V(G)$ until convergence. In each iteration (line 10-20), we find vertices v' with $\mathsf{active}(v')$ = true and $\overline{\mathsf{core}}(v')$ not being increased (line 11). For each such vertex v', we increase $\overline{\mathsf{core}}(v')$ by 1 (line 13), and load $N(v')$ from disk. Since $\overline{\mathsf{core}}(v')$ is changed, we need to compute $cnt(v')$ (line 15) and update the cnt values for the neighbors of v' (line 16-17). In line 18-20, we set $\mathsf{active}(u')$ to be true for all the neighbors u' of v' if u' is a possible candidate. Now we can guarantee that $\overline{\mathsf{core}}(v')$ is an upper bound of the new core number of v'. Therefore, we invoke line 3-12 of Algorithm 12 to compute the core numbers of all vertices in the graph (line 21).

Example 9. *Suppose after deleting edge (v_0, v_1) from the graph G (Fig. 4.1) in*

Iteration\v	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
Old Value	2	2	2	2	2	2	2	2	1
Iteration 1.1	2	2	2	2	3	3	3	3	1
Iteration 1.2	2	2	3	3	3	3	3	3	1
Iteration 1.3	3	3	3	3	3	3	3	3	1
Iteration 2.1	2	2	2	3	3	3	3	2	1

Table 4.4: ILLUSTRATION OF SemiInsert (INSERT (v_4, v_6))

Example 8, we insert a new edge (v_4, v_6) into G. The process to compute the new core numbers of vertices in G is shown in Table 4.4. Here, we use 3 iterations 1.1, 1.2, and 1.3 to compute the candidate vertices, and use 1 iteration 2.1 to compute the new core numbers. In iteration 1.1, when v_4 is computed, it triggers its smaller neighbors v_2 and v_3 to be computed in the next iteration and triggers its larger neighbor v_5 to be computed in the current iteration. The total number of vertex computations is 12.

Algorithm Analysis. The first phase of Algorithm 14 (line 8-20) locates all candidate vertices in V_c. The second phase of Algorithm 14 (line 21) performs the same operations as Algorithm 12. Let l_1 and l_2 be the number of iterations in the first and second phases respectively, we have:

Theorem 12. *Algorithm 14 requires $O(n)$ memory. The I/O and CPU time complexities of Algorithm 14 are $O(\frac{(l_1+l_2)\cdot(m+n)}{B})$ and$O(\sum_{w\in V_c} deg(w)+(l_1+l_2)\cdot n)$ respectively.*

Here, both l_1 and l_2 are bounded by $|V_c|$, which is much smaller than n in practice.

4.5.3 Optimization for Edge Insertion

The Rationale. Algorithm 14 handles an edge insertion using two phases. In phase 1, we compute a superset V_c of vertices whose core numbers will be updated, and we increase the core numbers for all vertices in V_c by 1. In phase 2, we compute the core numbers of all vertices using Algorithm 12. One problem of Algorithm 14 is that the size of V_c can be very large, which may result in a large number of vertex computations and I/Os in both phase 1 and phase 2 of Algorithm 14. Therefore, it is crucial to reduce the size of V_c.

Now, suppose and edge (u, v) is inserted into the graph G; $cnt(u)$ and $cnt(v)$ are updated accordingly; and $\overline{\mathsf{core}}(w)$ values for all $w \in V(G)$ have not been updated . Without loss of generality, we assume that $\overline{\mathsf{core}}(u) < \overline{\mathsf{core}}(v)$ and let $c_{old} = \overline{\mathsf{core}}(u)$. Let V_c be the set of candidate vertices computed in Algorithm 14, i.e., V_c consists of all vertices that are reachable from u via a path that consists of vertices with $\overline{\mathsf{core}}$ equals c_{old}. Let $V_c^* \subseteq V_c$ be the set of vertices with $\overline{\mathsf{core}}$ updated to be $c_{old} + 1$ after inserting (u, v). We have the following lemmas:

Lemma 10. *(a) For $v' \in V_c \setminus V_c^*$, $cnt(v')$ keeps unchanged; (b) For $v' \in V_c^*$, $cnt(v')$ will not increase.*

Proof. This lemma can be easily verified according to Eq. 4.2 and Theorem 7. □

Lemma 11. *If $cnt(v') \geq c_{old} + 1$ for all $v' \in V_c$, then we have $V_c^* = V_c$.*

Proof. If we increase $\overline{\mathsf{core}}(v')$ by 1 for all $v' \in V_c$, it is easy to verify that $cnt(v')$ for all $v' \in V_c$ keep unchanged. Now suppose $cnt(v') \geq c_{old} + 1$ for all $v' \in V_c$, we can derive that the locality property in Theorem 9 holds for every $v' \in V(G)$. Therefore, the new $\overline{\mathsf{core}}(v')$ is the core number of v' for every $v' \in V(G)$. This indicates that $V_c^* = V_c$. □

Lemma 12. *For any $v' \in V_c$, if $v' \in V_c^*$, then we have $cnt(v') \geq c_{old} + 1$.*

Proof. Since $v' \in V_c^*$, we know that the new $cnt(v')$ is no smaller than $c_{old}+1$. According to Lemma 10 (b), the original $cnt(v')$ is also no smaller than $c_{old}+1$, since $cnt(v')$ will not increase. Therefore, the lemma holds. □

Theorem 13. *For each $v' \in V_c$, we define $cnt^*(v')$ as:*

$$cnt^*(v') = |\{u' \in N(v') \mid \overline{\mathsf{core}}(u') > c_{old} \text{ or } u' \in V_c^*\}| \tag{4.3}$$

We have:
(a) If $v' \in V_c^*$, then the updated $cnt(v') = cnt^*(v')$; and
(b) $v' \in V_c^* \Leftrightarrow cnt^*(v') \geq c_{old}+1$.

Proof. For (a): for all $v' \in V_c^*$, since $\overline{\mathsf{core}}(v')$ will become $c_{old}+1$, all vertices $u' \in V_c \setminus V_c^*$ will not contribute to $cnt(v')$ according to Eq. 4.2. Therefore, (a) holds.
For (b): $\Rightarrow$ can be derived according to (a). Now we prove $\Leftarrow$. Suppose $cnt^*(v') \geq c_{old}+1$, to prove $v' \in V_c^*$, we prove that if we increase $\overline{\mathsf{core}}(u')$ to $c_{old}+1$ for all $u' \in V_c$ and apply Algorithm 12, then $\overline{\mathsf{core}}(v')$ will keep to be $c_{old}+1$ after convergence. Note that for all vertices $u' \in V_c^*$ and $u' \in N(v')$, $\overline{\mathsf{core}}(u')$ will keep to be $c_{old}+1$ and will contribute to $cnt(v')$, and all vertices $u' \in N(v')$ with $\overline{\mathsf{core}}(u') > c_{old}$ will also contribute to $cnt(v')$. According to Eq. 4.3, we have $cnt(v') \geq cnt^*(v') \geq c_{old}+1$. Therefore, $\overline{\mathsf{core}}(v')$ will never decrease according to Lemma 9. This indicates that $v' \in V_c^*$. □

According to Theorem 13 (b), $cnt^*(v')$ can be defined using the following recursive equation:

$$cnt^*(v') = |\{u' \in N(v') \mid \overline{\mathsf{core}}(u') > c_{old} \text{ or}$$

$$(\overline{\mathsf{core}}(u') = c_{old} \text{ and } cnt^*(u') \geq c_{old} + 1)\}| \quad (4.4)$$

To compute $cnt^*(v')$ for all $v' \in V_c$, we can initialize $cnt^*(v')$ to be $cnt(v')$, and apply Eq. 4.4 iteratively on all $v' \in V_c$ until convergence. However, this algorithm needs to compute V_c first, which is inefficient. Note that according to Eq. 4.4 and Theorem 13 (b), we only care about those vertices u' with $cnt^*(u') \geq c_{old}+1$. Therefore, we do not need to compute the whole V_c by expanding from vertex u. Instead, for each expanded vertex u', if we guarantee that $cnt^*(u') < c_{old} + 1$, we do not need to expand u' further. In this way, the computational and I/O cost can be largely reduced.

Algorithm Design. Based on the above discussion, for each vertex $w \in V(G)$, we use $\mathsf{status}(w)$ to denote the status of vertex w during the processing of vertex expansion. Each vertex $w \in V(G)$ has the following four status ($\mathsf{status}(w)$):

- ϕ: w has not been expanded by other vertices.
- ?: w is expanded but $cnt^*(w)$ is not calculated.
- $\surd$: $cnt^*(w)$ is calculated with $cnt^*(w) \geq c_{old} + 1$.
- $\times$: $cnt^*(w)$ is calculated with $cnt^*(w) < c_{old} + 1$.

With $\mathsf{status}(w)$ and according to Theorem 13 (a) and Lemma 10 (a), we can reuse $cnt^*(w)$ to calculate $cnt(w)$ for each $w \in V(G)$. That is, if $\mathsf{status}(w) = \surd$, $cnt^*(w)$ can directly represent $cnt(w)$ according to Eq. 4.4, otherwise, if $\mathsf{status}(w) = \times$, $cnt(w)$ is calculated using Eq. 4.2.

Our new algorithm $\mathsf{SemiInsert}^*$ for edge insertion is shown in Algorithm 15. The initialization phase is similar to that in Algorithm 14 (line 1). In line 2, we initialize $\mathsf{status}(w)$ to be ϕ except $\mathsf{status}(u)$ which is initialized to be ?. The algorithm iteratively update $\mathsf{status}(v')$, $\overline{\mathsf{core}}(v')$, and $cnt(v')$ for all $v' \in V(G)$. For each such v' to be checked in every iteration (line 4-26), we consider the

Algorithm 15 SemiInsert*(graph G on disk, edge (u, v))

```
1: line 1-5 of Algorithm 14;
2: status(w) ← (ϕ) for all w ∈ V(G); status(u) ← (?);
3: update ← true;
4: while update do
5:    update ← false;
6:    for v' ← v_1 to v_n do
7:       if status(v') = (?) then
8:          update ← true;
9:          load N(v') from disk;
10:         cnt(v') ← ComputeCnt*(N(v'), c_old);
11:         status(v') ← (√); core(v') ← c_old + 1;
12:         for each u' ∈ N(v') s.t. core(u') = c_old + 1 do
13:            cnt(u') ← cnt(u') + 1;
14:         if cnt(v') ≥ c_old + 1 then
15:            for each u' ∈ N(v') s.t. core(u') = c_old do
16:               if cnt(u') ≥ c_old + 1 and status(u') = (ϕ) then
17:                  status(u') ← (?);
18:      if status(v') = (√) and cnt(v') < c_old + 1 then
19:         update ← true;
20:         load N(v') from disk if not loaded;
21:         cnt(v') ← ComputeCnt*(N(v'), c_old);
22:         status(v') ← (⊗); core(v') ← c_old;
23:         for each u' ∈ N(v') s.t. core(u') = c_old + 1 do
24:            cnt(u') ← cnt(u') − 1;
25:         for each u' ∈ N(v') s.t. status(u') = (√) do
26:            cnt(u') ← cnt(u') − 1;
27: Procedure ComputeCnt*(N(v'), c_old)
28: s ← 0;
29: for each u' ∈ N(v') do
30:    if core(u') > c_old or (core(u') = c_old and cnt(u') ≥ c_old + 1 and status(u') ≠ (⊗)) then
       s ← s + 1;
31: return s;
```

following status transitions:

• *From* (?) *to* (√) *(line 7-13):* If status$(v') =$ (?) (line 7), we load $N(v')$ from disk (line 9) and compute $cnt(v')$ using Eq. 4.4 by invoking ComputeCnt$^*(N(v'), c_{old})$ which is shown in line 27-31. Compared to Eq. 4.4, we add a new condition for $u' \in N(v')$: status$(u') \neq$ (⊗) (line 30). This is because for vertex u' with status$(u') =$ (⊗), it is computed using Eq. 4.2 other than Eq. 4.4, and it cannot contribute to $cnt(v')$. After computing $cnt(v')$, in line 11, we set status(v') to be

$\surd$ and increase $\overline{\mathsf{core}}(v')$ to be $c_{old}+1$. Since $\overline{\mathsf{core}}(v')$ is increased to be $c_{old}+1$, we need to increase $cnt(u')$ for all neighbor u' of v' with $\overline{\mathsf{core}}(u') = c_{old}+1$ (line 12-13).

• *From* ϕ *to* ? *(line 14-17):* After setting v' to be ?, if $cnt(v') \geq c_{old}+1$, v' will not set to be $\otimes$ in this iteration. In this case (line 14), we can expand v'. That is, for all neighbors u' of v' with $\overline{\mathsf{core}}(u') = c_{old}$ (line 15), if $cnt(u') \geq c_{old}+1$ (refer to Lemma 12) and u' has not be expanded ($\mathsf{status}(u') = \phi$), we set $\mathsf{status}(u')$ to be ? so that u' can be expanded.

• *From* $\surd$ *to* $\otimes$ *(line 18-26):* If $\mathsf{status}(v')$ is $\surd$ and $cnt(v') < c_{old}+1$, we need to change the status of v' (line 18). Here, in line 20, we load $N(v')$ from disk if it is not loaded in line 9. In line 21, we compute $cnt(v')$ using Eq. 4.2. In line 22, we set $\mathsf{status}(v')$ to be $\otimes$, and update $\overline{\mathsf{core}}(v')$ to be c_{old} according to Lemma 10 (a). Since $\overline{\mathsf{core}}(v')$ is changed from $c_{old}+1$ to c_{old}, for all neighbors u' of v' with $\overline{\mathsf{core}}(u') = c_{old}+1$, we need to decrease $cnt(u')$ (line 23-24). In addition, according to Eq. 4.4, the status change from $\surd$ to $\otimes$ for v' will trigger each neighbor u' of v' to decrease its $cnt(u')$ if $\mathsf{status}(u') = \surd$ (line 25-26).

Iteration\v	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
Old Value	2	2	2	2	2	2	2	2	1
Iteration 1	ϕ	ϕ	?	?	$\surd$	$\surd$	$\surd$	ϕ	ϕ
Iteration 2	ϕ	ϕ	$\otimes$	$\surd$	$\surd$	$\surd$	$\surd$	ϕ	ϕ
New Value	2	2	2	3	3	3	3	2	1

Table 4.5: ILLUSTRATION OF $\mathsf{SemiInsert}^*$ (INSERT (v_4, v_6))

Example 10. *Suppose after deleting edge* (v_0, v_1) *from graph* G *(Fig. 4.1) in Example 8, we insert edge* (v_4, v_6) *into* G*. The process to update the status of vertices in each iteration is shown in Table 4.5. In iteration* 1*, when we check* v_4*, we update* $\mathsf{status}(v_4)$ *from* ? *to be* $\surd$*, and update the status of its neighbors* (v_2,

v_3, v_5, and v_6) to be ?. In iteration 2, for v_2 with status ?, we can calculate that $cnt(v_2) = 2 < c_{old} + 1 = 3$. Therefore, we set status(v_2) *to be ⊗, and decrease $cnt(v_4)$ accordingly. The cells involving a vertex computation are marked grey. Totally 2 iterations are needed. The four vertices v_3, v_4, v_5, and v_6 with* status *being √ have their core numbers updated. Compared to Example 9, we decrease the number of vertex computations from 12 to 5.*

Algorithm Analysis. Theoretically, Algorithm 15 is bounded by the same space, CPU time and I/O complexities as Algorithm 14. However, compared to Algorithm 14 that requires two phases to update the core numbers, Algorithm 15 requires only one phase and the number of candidate vertices is largely reduced in Algorithm 15.

4.6 I/O Efficient Degeneracy Ordering

In this section, we study the problem of degeneracy ordering, which is closely related to the core decomposition problem. Given an arbitrary total order $f()$ for the vertices in graph G, let $\vec{deg}(u)$ be the backward-degree, i.e., $\vec{deg}(u) = |\{v|v \in N(u), f(v) > f(u)\}|$. The degeneracy of the order is the maximum $\vec{deg}(u)$ for all u. The definition of graph degeneracy is given as follows:

Definition 13. ***(Graph Degeneracy [32])*** *The degeneracy of a graph G, denoted by $d(G)$, is the minimum degeneracy of any total order of vertices in G.*

Definition 14. ***(Degeneracy Order [32])*** *A total order of vertices in G is a degeneracy order if $\forall u \in V, \vec{deg}(u) \leq d(G)$.*

Example 11. *Fig. 4.3 shows a degeneracy order of vertices in graph G of Fig. 4.1 from left to right. The corresponding backward-degree of each vertex is given in the table. The degeneracy of G is 3. We consider vertex v_2. The neighbors*

of vertex v_2 in G are v_0, v_1, v_3 and v_4. In the shown order, vertex v_2 has 2 backward-neighbors, v_3 and v_4. Thus the backward-degree here for v_2 is 2.

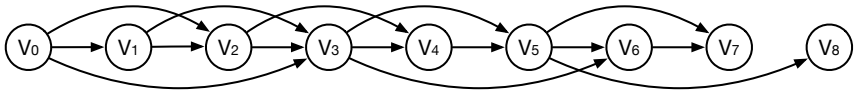


Figure 4.3: A degeneracy order of vertices in Fig. 4.3

Order	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
Backward-Deg	3	2	2	3	1	3	1	0	0

An in-memory algorithm for the computation of the degeneracy order is proposed in [56]. The algorithm is similar to the in-memory algorithm for core decomposition [9]. Specifically, it iteratively removes the vertex with minimum degree in the graph until all vertices are removed. After removing a vertex, the degrees of other vertices are updated accordingly. The sequence of the removed vertices forms the degeneracy order, and the maximum backward-degree of vertices in the sequence is the degeneracy of graph G. It costs $O(m)$ time and $O(m)$ space. Since we follow the same process of in-memory core decomposition to compute the degeneracy order, we can derive that the graph degeneracy is equal to the max core value, i.e. k_{max}, in the graph [32]:

$$d(G) = k_{max} \tag{4.5}$$

We handle the scenario that the graph is too big to load in the memory. Note that the in-memory core decomposition algorithm [9] follows a bottom-up strategy and the sequence of removed vertices naturally forms a degeneracy order according to [56]. By contrast, in Algorithm 12, we follow a top-down strategy

which iteratively updates the core numbers of all vertices until convergence. Thus we cannot directly obtain the degeneracy order as a byproduct of Algorithm 12.

4.6.1 Degeneracy Order Computation

We aim to compute the degeneracy order on big graphs. A straightforward semi-external algorithm can be easily obtained by following the idea of [56], which iteratively removes the vertices with the minimum degree and terminates once all vertices are removed. The algorithm is given in Algorithm 16.

Algorithm 16 Dorder(graph G on disk)

```
1: 𝒟 ← ∅, Deg(v) ← deg(v) for all v ∈ V(G); d_min ← −1;
2: while |𝒟| < |V| do
3:     d_min ← max(d_min, min_{u∈V\𝒟} Deg(u));
4:     for v ← v_1 to v_n do
5:         if v ∈ 𝒟 or Deg(v) > d_min then continue;
6:         𝒟.append(v);
7:         load N(v) from disk;
8:         for all u ∈ N(v) do Deg(u) ← Deg(u) − 1;
9: return 𝒟;
```

The Rationale. Obviously, Algorithm 16 is inefficient since in each iteration it can only process the vertices with the minimum degree. Note that by performing the semi-external core decomposition algorithm, we can compute the degeneracy of the graph as $d(G) = \max_{v \in V(G)} core(v)$. With $d(G)$, we can remove more vertices then just removing the vertices with the minimum degree in each iteration. Specifically, according to Definiton 14, we remove all vertices with degree no larger than $d(G)$ in each iteration, update the degree of the remaining vertices, and terminate once all vertices are removed. Since the degree of each removed vertex is less then $d(G)$, the sequence of the removed vertices forms the degeneracy order.

Algorithm Design. The algorithm Dorder* is shown in Algorithm 17. We invoke the SemiCore* algorithm for graph G in advance to calculate the core

Algorithm 17 Dorder*(graph G on disk)

```
1: invoke Algorithm 12 to compute core(v) for all v ∈ V(G);
2: d(G) ← max_{v∈V(G)} core(v);
3: D ← ∅;
4: Deg(v) ← deg(v) for all v ∈ V(G);
5: while |D| < |V| do
6:    for v ← v_1 to v_n do
7:       if v ∈ D or Deg(v) > d(G) then continue;
8:       D.append(v);
9:       load N(v) from disk;
10:      for all u ∈ N(v) do Deg(u) ← Deg(u) − 1;
11: return D;
```

number of each vertex (line 1). We select the maximum core number as $d(G)$ (line 2). We use an ordered list $\mathcal{D}$ to keep the degeneracy order of all vertices (line 3), and we use $Deg(u)$ to maintain the degree of each vertex $u \in V(G)$ (line 4). In each iteration, we sequentially scan the vertices. For each vertex u such that u is not added to $\mathcal{D}$ and $Deg(u) \leq d$, we append it to the end of the degeneracy order $\mathcal{D}$ (line 8) and decrease the degree of all neighbors of u by 1 (line 10). The algorithm terminates once all vertices are added to $\mathcal{D}$.

Algorithm Analysis. Let l be the number of iterations of Algorithm 17, we have the following theorem:

Theorem 14. *The space, I/O, and time complexities of Algorithm 17 are* $O(n)$, $O(\frac{l \cdot (m+n)}{B})$, *and* $O(m)$ *respectively.*

Here, l can be bounded by n in the worst case. However, it is much smaller in practice. In our experiments, the number of iterations is less than 5 on most of our tested datasets. Note that the cost to invoke Algorithm 12 is not included in Theorem 14.

4.6.2 Degeneracy Order Maintenance

In this section, we consider the scenario when the graph updates dynamically. We aim to design a semi-external algorithm to maintain the degeneracy order incrementally. We only focus on edge update since the vertex update can be implemented by a series of edge updates. Note that a special case here is that graph degeneracy $d(G)$ changes when the graph updates. We just recompute the degeneracy order for this case. Thus for each inserted or removed edge, we first invoke our algorithm SemiInsert* to obtain the maximum core number k_{max} and check whether $d(G)$ changes using Eq. 4.5. In the following, we assume that the graph degeneracy does not change when the graph updates.

The Rationale. The main challenge of degeneracy order maintenance is that we need to maintain a total order of vertices in the graph every time the graph updates. From Algorithm 17, we observe that in each iteration, there are multiple vertices u with $Deg(u) < d(G)$. It is easy to see that we can remove these vertices in an arbitrary order in the iteration to generate the degeneracy order. We can simply put these vertices or any subset of these vertices in a certain level and thus divide the vertices of the graph into several levels. With the levels, instead of maintaining the total order of vertices, we can just maintain the vertex levels to update the degeneracy order.

Level-Index. We propose a Level-Index which contains the following two components:

(i) The *level-value* for each vertex u is a value $\mathcal{L}(u)$ satisfying:

$$|\{v \in N(u) | \mathcal{L}(v) \geq \mathcal{L}(u)\}| \leq d(G) \tag{4.6}$$

(ii) The *upper-degree* for each vertex u is the number of its neighbors v whose

level-value is not less than that of u:

$$Deg_{\mathcal{L}}(u) = |\{v \in N(u) | \mathcal{L}(v) \geq \mathcal{L}(u)\}| \tag{4.7}$$

Example 12. *An example of* Level-Index *for graph G in Fig. 4.1 is given in Fig. 4.4. There exists only 2 levels for all vertices in G. The edges connecting two vertices in the same level are shown in undirected lines and those connecting the vertices in different levels are shown in directed lines. The level-value and upper-degree for all vertices are presented in the table. We can see that the upper-degree of any vertex is not larger than $d(G) = 3$. We consider the vertex v_0. There are 3 neighbors of v_0 which has 3 neighbors, v_1, v_2 and v_3. Given $\mathcal{L}(v_0) = \mathcal{L}(v_1) = 1$ and $\mathcal{L}(v_2) = \mathcal{L}(v_3) = 2$, we have $Deg_{\mathcal{L}}(v_0) = 3$. For the vertex v_2, there are 4 neighbors, v_0, v_1, v_3 and v_4. Since $\mathcal{L}(v_0) = \mathcal{L}(v_1) = \mathcal{L}(v_4) = 1$ and $\mathcal{L}(v_2) = \mathcal{L}(v_3) = 2$, we have $Deg_{\mathcal{L}}(v_2) = 1$.*

Given a Level-Index, we can obtain a degeneracy order as follows. We sort all vertices in non-decreasing order of their level values, and for those vertex with the same level value, we sort them in an arbitrary order. It is easy to see that any vertex order generated in this way is a degeneracy order. Therefore, instead of maintaining the degeneracy order, we only need to maintain the Level-Index. In the following, we first show how to construct the Level-Index, and then introduce how to maintain the Level-Index when an edge is removed or inserted.

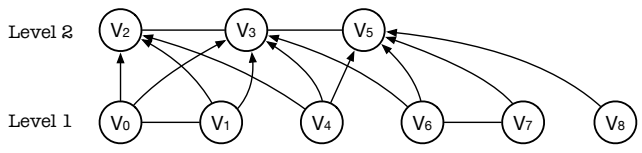


Figure 4.4: A Level-Index for graph G in Fig. 4.1

Level-Index	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
$\mathcal{L}$	1	1	2	2	1	2	1	1	1
$Deg_{\mathcal{L}}$	3	3	1	2	3	1	3	2	1

Algorithm Design for Level-Index Construction. The algorithm to construct the Level-Index is given in Algorithm 18. We follow the semi-external model. The $\mathcal{L}$ and $Deg_{\mathcal{L}}$ for all vertices are initialized by -1 (line 1) and original degree (line 2) respectively. The *level* is initialized by -1 (line 3). In each iteration, we assign the level of all vertices whose degree are not larger than the $d(G)$ by current *level* (line 6 and 7). The current degree of them are their corresponding *upper-degree*. Then we remove such vertices and update the degree of their neighbors (line 9-12). We increase the *level* (line 13) and move to next iterations. The algorithm terminates until the *level-value*s of all vertices in G are assigned.

Algorithm 18 Level-IndexConstruction$(G, \mathcal{L}, Deg_{\mathcal{L}})$

```
1: L(v) ← −1 for all v ∈ V(G);
2: Deg_L(v) ← deg(v) for all v ∈ V(G);
3: level ← 1;
4: while there exists a vertex u such that L(u) = −1 do
5:     for v ← v_1 to v_n do
6:         if L(v) ≠ −1 or Deg_L(v) > d(G) then continue;
7:         L(v) ← level;
8:     for v ← v_1 to v_n do
9:         if L(v) ≠ level then continue;
10:        load N(v) from disk;
11:        for u ∈ N(v) do
12:            if L(v) = −1 then Deg_L(u) ← Deg_L(u) − 1;
13:    level ← level + 1;
```

Based on Eq. 4.6 and Eq. 4.7, we can derive the following sufficient and necessary condition for $\mathcal{L}$ to be updated.

Lemma 13. *For each vertex $v' \in V(G)$, $\mathcal{L}(v')$ is updated if and only if $Deg_{\mathcal{L}}(v') > d(G)$.*

Proof. We first prove $\Leftarrow$: Suppose $Deg_{\mathcal{L}}(v') > d(G)$, we have $deg(v') > d(G)$ after removing all vertices u with $\mathcal{L}(u') < \mathcal{L}(v')$. This contradicts the definition of *level-value* in Level-Index. Therefore, $\mathcal{L}(v')$ needs to be updated.

Next, we prove $\Rightarrow$: Suppose $Deg_{\mathcal{L}}(v')$ needs to be updated, according to the definition of Level-Index, either $|\{u' \in N(v') | \mathcal{L}(u') \geq \mathcal{L}(v')\}| > d(G)$ or there is a smaller l s.t. $\mathcal{L}(v') = l$ and $|\{u' \in N(v') | \mathcal{L}(u') \geq \mathcal{L}(v')\}| \leq d(G)$. The latter is impossible since $\mathcal{L}(v')$ will never decrease when $d(G)$ does not change. Therefore, we have $Deg_{\mathcal{L}}(v') > d(G)$. □

Algorithm Analysis. The space, CPU, and I/O complexities of Algorithm 18 are $O(n), O(m)$ and $O(\frac{l \cdot (m+n)}{B})$ respectively, where l is the number of levels.

Algorithm Design for Edge Deletion. We consider removing an edge. Based on Lemma 13, after removing an edge (u, v), if $d(G)$ is unchanged, we only need to update $Deg_{\mathcal{L}}(u)$ and $Deg_{\mathcal{L}}(v)$ according to Eq. 4.7.

Algorithm Design for Edge Insertion. We consider inserting an edge (u, v). We first update $Deg_{\mathcal{L}}(u)$ and $Deg_{\mathcal{L}}(v)$ based on Eq. 4.7. According to Lemma 13, we need to find a new *level-value* for any vertex v' in G if $Deg_{\mathcal{L}}(v') > d(G)$. Note that the raise of *level-value* of a vertex v' will increase the *upper-degree* of its neighbors. This may lead to some vertices $u' \in N(v')$ such that $Deg_{\mathcal{L}}(u') > d(G)$. To minimize such influence, we need to find a minimum new *level-value* for vertex v'. The pseudocode for finding a minimum new *level-value* for a vertex v' is given in Procedure LocalLevel() of Algorithm 19. ml is the maximum *level-value* in the graph (line 17). Given a vertex v', we use an array num to record the number of each *level-value* for its neighbors (line 18-19). We initialize the new *level-value* $\mathcal{L}_{new}$ for vertex v' to be $ml+1$ and use cd to calculate the *upper-degree* for the level $\mathcal{L}_{new}$. We find the minimum $\mathcal{L}_{new}$ satisfying $cd \leq d(G)$ (line 21-24) and return $\mathcal{L}_{new}$ as the new $\mathcal{L}(v')$.

The overall algorithm for edge insertion is given in Algorithm 19. SemiInsert*

Algorithm 19 DInsert$^*(G, \mathcal{L}, Deg_{\mathcal{L}}, edge(u, v))$

```
1: invoke SemiInsert*(G, (u, v)) to update core number;
2: if d(G) changes (based on Eq. 4.5) then recompute the index and return;
3: if L(u) ≤ L(v) then Deg_L(u) ← Deg_L(u) + 1;
4: if L(v) ≤ L(u) then Deg_L(v) ← Deg_L(v) + 1;
5: if Deg_L(u) ≤ d(G) and Deg_L(v) ≤ d(G) then return;
6: update ← true;
7: while update do
8:     update ← false;
9:     for v' ← v_1 to v_n s.t. Deg_L(v') > d(G) do
10:        update ← true;
11:        load N(v') from disk;
12:        L_old ← L(v');
13:        L(v') ← LocalLevel(N(v'));
14:        Deg_L(v') ← ComputeUDeg(N(v'));
15:        UpdateNbrUDeg(N(v'), L_old, L(v'));
16: Procedure LocalLevel(N(v'))
17: ml ← max_{u'∈V} L(u')
18: num(i) ← 0 for all 1 ≤ i ≤ ml;
19: for all u' ∈ N(v') do num(L(u')) ← num(L(u')) + 1;
20: cd ← 0; L_new ← ml + 1;
21: for i ← ml downto 1 do
22:     cd ← cd + num(i);
23:     if cd > d(G) then break;
24:     L_new ← i;
25: return L_new;
26: Procedure ComputeUDeg(N(v'))
27: ud ← 0;
28: for each u' ∈ N(v') do
29:     if L(u') ≥ L(v') then ud ← ud + 1;
30: return ud;
31: Procedure UpdateNbrUDeg(N(v'), L_old, L(v'))
32: for each u' ∈ N(v') do
33:     if L_old < L(u') ≤ L(v') then Deg_L(u') ← Deg_L(u') + 1;
```

is firstly invoked to check whether $d(G)$ changes. The $Deg_{\mathcal{L}}$ of u and v are updated according to Eq. 4.7 (line 3-5). Then we adopt Lemma 13 and focus on the vertex v' with $Deg_{\mathcal{L}}(v') > d(G)$ in line 9. In line 13, we invoke LocalLevel (line 16-25) to calculate the new *level-value* $\mathcal{L}(v')$. With new $\mathcal{L}(v')$, we invoke ComputeUDeg (line 26-30) in line 14 to calculate the corresponding $Deg_{\mathcal{L}}(v')$ according to Eq. 4.7 (line 29). In line 15, we invoke UpdateNbrUDeg (line 31-33)

to update $Deg_{\mathcal{L}}$ of the neighbors of v'. Based on Eq. 4.7, only those neighbors u' with $\mathcal{L}(u')$ falling in the range $(\mathcal{L}_{old}, \mathcal{L}(v')]$ will have $Deg_{\mathcal{L}}(u')$ increase by 1 (line 33).

Iteration\v	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
Init	1	1	2	2	1	2	1	1	1
Iteration 1	2	1	2	2	1	2	2	1	1
Iteration 2	2	1	2	3	1	2	2	1	1

Table 4.6: Illustration of DInsert* (insert (v_0, v_6))

Example 13. *Table 4.6 gives an example that we insert an edge (v_0, v_6) into G of Fig. 4.1. The initial $\mathcal{L}$ of each vertex is given in the second row. We know the degeneracy $d(G)$ of G is 3. After inserting the edge (v_0, v_6), we have $Deg_{\mathcal{L}}(v_0) = 4$ and $Deg_{\mathcal{L}}(v_0) > d(G)$. Thus we update $\mathcal{L}(v_0)$ from 1 to 2. Similarly, we update $\mathcal{L}(v_6)$ from 1 to 2. After the first iteration, we only have $Deg_{\mathcal{L}}(v_3) = 4 > d(G)$. We update $\mathcal{L}(v_3)$ to 3 in the second iteration. The algorithm terminates after the second iteration.*

Algorithm Analysis. Let l be the number of iterations in Algorithm 19. The space, I/O, and time complexities of Algorithm 19 are $O(n)$, $O(\frac{l \cdot (m+n)}{B})$, and $O(l \cdot (m+n))$ respectively. Here, l is very small in practice. For example, in our experiments, l is not larger than 10 in all the tested datasets.

4.7 Performance Studies

In this section, we experimentally evaluate the performance of our proposed algorithms for both k-core and degeneracy order. Subsection 4.7.1 compares our solutions with state-of-the-art algorithms; Subsection 4.7.2 shows the efficiency

of our maintenance algorithm; We report the scalability of these two algorithms in Subsection 4.7.5; Subsection 4.7.3 reveals the efficiency of degeneracy ordering algorithm and Subsection 4.7.4 focuses on the degeneracy order maintenance.

All algorithms are implemented in C++, using gcc complier at -O3 optimization level. All the experiments are performed under a Linux operating system running on a machine with an Intel Xeon 3.4GHz CPU, 16GB RAM and 7200 RPM SATA Hard Drives (2TB). The cost of each algorithm is measured as the amount of wall-clock time elapsed during the program's execution. We adhere to standard external memory model for I/O statistics [1].

Datasets. We use two groups of datasets to demonstrate the efficiency of our semi-external algorithm. Group one consists of six graphs with relatively small size: DBLP, Youtube, WIKI, CPT, LiveJournal and Orkut. Group two consists of six big graphs: Webbase, IT, Twitter, SK, UK-2002 and Clueweb. The detailed information for the 12 datasets is displayed in Table 4.7. Here, the CSR size is the size of the graph represented using the binary Compressed Sparse Row (CSR) format.

In group one (small graphs), DBLP is a co-authorship network of DBLP. Youtube is a social network based on the user friendship in Youtube. WIKI is a network containing all the users and discussion from the inception of Wikipedia till January 2008. CPT is a citation graph that consists in all citations made by patents granted between 1975 and 1999. LiveJournal (LiveJournal) is a free online blogging community. Orkut is a free online social network.

In group two (big graphs), IT is a fairly large crawl of the `.it` domain. Twitter is a social network collected from Twitter where vertices are users and edges follow tweet transmission. Webbase is a graph obtained from the 2001 crawl performed by the WebBase crawler. SK is a graph obtained from a 2005 crawl of the `.sk` domain. UK-2002 is a graph gathering a snapshot of about 100

Datasets	$\|V\|$	$\|E\|$	density	k_{max}	CSR size
DBLP	317K	1,049K	3.31	113	9.7 MB
Youtube	1,134K	2,987K	2.63	51	28.4 MB
WIKI	2,394K	5,021K	2.10	131	49.7 MB
CPT	3,774K	16,518K	4.38	64	147.2 MB
LiveJournal	3,997K	34,681K	8.67	360	293.4 MB
Orkut	3,072K	117,185K	38.14	253	949.8 MB
IT	41,291K	1,150,725K	27.86	3224	9.4 GB
Twitter	41,652K	1,468,365K	35.25	2488	11.9 GB
Webbase	118,142K	1,019,903K	8.63	1506	8.6 GB
SK	50,636K	1,949,412K	38.49	4510	15.8 GB
UK-2002	105,896K	3,738,733K	35.30	5704	30.3 GB
Clueweb	978,408K	42,574,107K	43.51	4244	344.5 GB

Table 4.7: Network statistics (1K = 10^3)

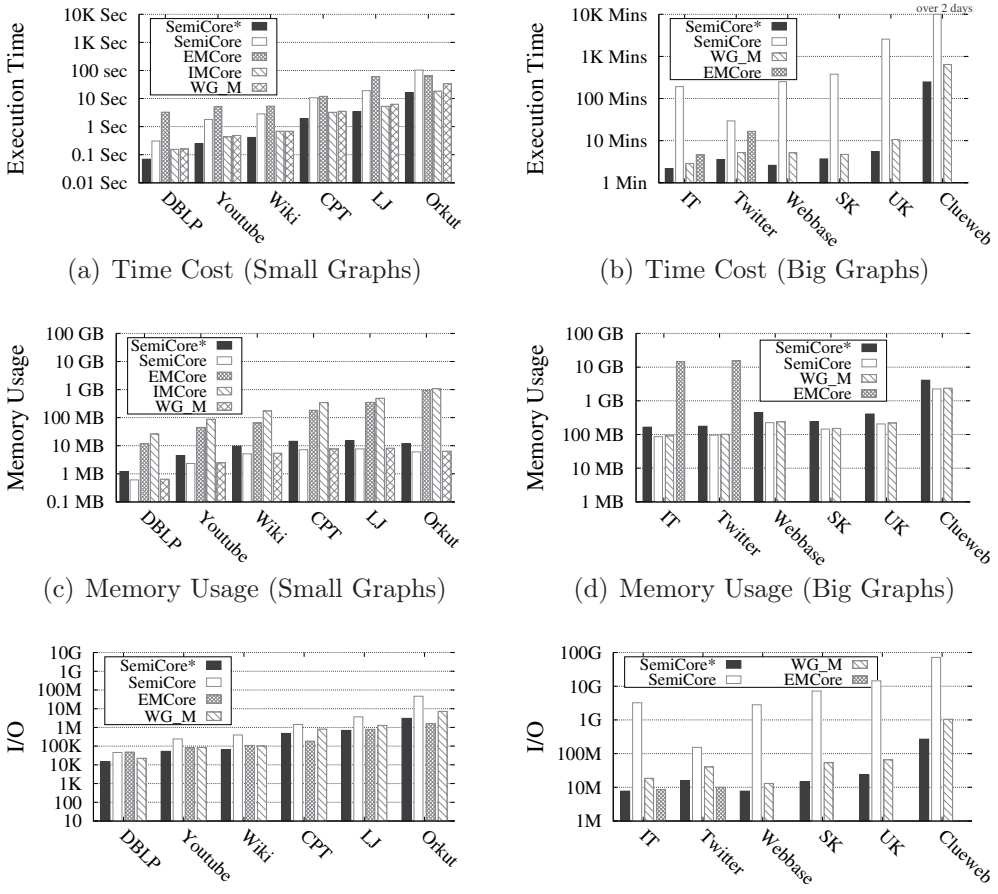
million pages for the DELIS project in May 2007. Finally, Clueweb is a web graph underlying the ClueWeb12 dataset. All datasets can be downloaded from SNAP (http://snap.stanford.edu/) and LAW (http://law.di.unimi.it/).

4.7.1 Core Decomposition

To explicitly reveal the performance of our core decomposition algorithms, we select the core decomposition algorithm EMCore [19] and the in-memory algorithm [9], denoted by IMCore as comparisons. We also report the results of algorithm WG_M [44], which is a parallel work for core decomposition in large graphs. WG_M also follows the semi-external model. More details of WG_M are given in Chapter 2.

Small Graphs. As shown in Fig. 4.5 (a), the total running time of SemiCore* is the fastest. It is 10 times faster than that of the EMCore on average. It is remarkable that SemiCore* can be even faster than the in-memory algorithm IMCore. Fig. 4.5 (c) shows that algorithm SemiCore* requires less memory than

EMCore and IMCore. Among all algorithms, SemiCore uses the least amount of memory since it does not rely on the cntnumbers for all vertices. By contrast, EMCore uses a large amount of memory. Especially in Orkut and CPT, EMCore uses almost the same memory size as IMCore. Fig. 4.5 (e) shows the I/O consumption of all algorithms except IMCore. SemiCore* and EMCore usually use the least amount of I/Os. However, due to the simple read-only data access of SemiCore*, SemiCore* is much more efficient than EMCore (refer to Fig. 4.5 (a)).



(a) Time Cost (Small Graphs) (b) Time Cost (Big Graphs)

(c) Memory Usage (Small Graphs) (d) Memory Usage (Big Graphs)

(e) I/Os (Small Graphs) (f) I/Os (Big Graphs)

Figure 4.5: Core decomposition on different datasets

Big Graphs. We report the performance of our algorithms on big graphs in

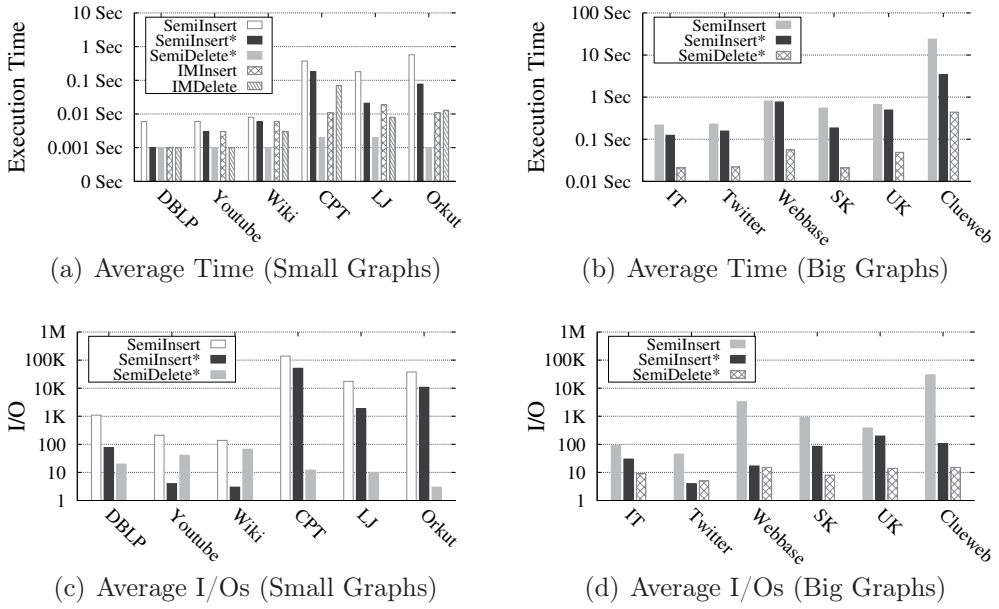
Fig. 4.5 (b), (d), and (f). Note that algorithm EMCore can only successfully process IT and Twitter using the machine with 16GB RAM. The largest dataset Clueweb contains nearly 1 billion vertices and 42.6 billion edges. We can see from Fig. 4.5 (a) that SemiCore* can process all datasets within 10 minutes except Clueweb. In Fig. 4.5, we can see that SemiCore* totally costs less than 4.2 GB memory to process the largest dataset Clueweb. This result demonstrates that our algorithm can be deploy in any commercial machine to process big graph data. Fig. 4.5 (f) further reveals the advance of optimization in terms of I/O cost, since SemiCore* spends much less I/Os than SemiCore and WG_M in all datasets.

4.7.2 Core Maintenance

We test the performance of our maintenance algorithms SemiInsert, SemiInsert*, and SemiDelete*. Since there are not previous algorithms for core maintenance in big graphs, we add the state-of-the-art streaming in-memory algorithms [68] for comparison, which are denoted by IMInsert and IMDelete. In the machine with 16GB RAM, IMInsert and IMDelete can only process the graph updates for the small graphs. We randomly select 100 distinct existing edges in the graph for each test. To test the performance of edge deletion, we remove the 100 edges from the graph one by one and take the average processing time and I/Os. To test the performance of edge insertion, after the 100 edges are removed, we insert them into the graph one by one and take the average processing time and I/Os. The experimental results are reported in Fig. 4.6.

From Fig. 4.6, we can see that SemiDelete* is more efficient than SemiInsert* in both processing time and I/Os for all datasets. This is because SemiDelete* simply follows SemiCore* and does not rely on the calculation of other new graph properties. From Fig. 4.6 (a), we can find that SemiDelete* is even faster than

IMDelete for edge deletion. This is due to the simple structures and data access model used in SemiDelete*. SemiInsert* outperforms SemiInsert in both processing time and I/Os for all datasets.



(a) Average Time (Small Graphs)

(b) Average Time (Big Graphs)

(c) Average I/Os (Small Graphs)

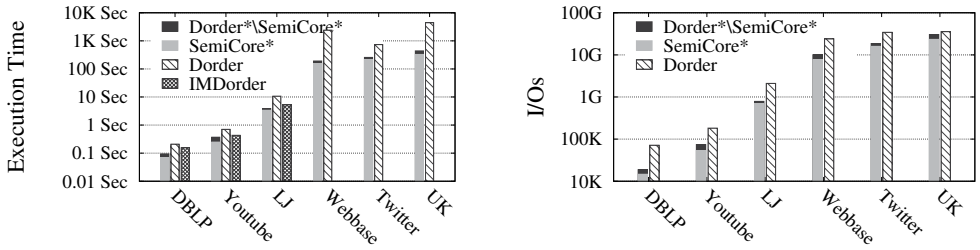
(d) Average I/Os (Big Graphs)

Figure 4.6: Core maintenance on different datasets

4.7.3 Degeneracy Order Computation

We present the performance of the algorithms Dorder and Dorder* for computing the degeneracy order. To explicitly reveal the efficiency of Dorder*, we show the results on three small graphs (DBLP, Youtube and LiveJournal) and three big graphs (Webbase, Twitter and UK-2002). We also report the results of the in-memory algorithm IMDorder [56] on three small graphs for comparison.

The experimental results are presented in Fig. 4.7. We show both the time cost and the I/O cost of Dorder* and Dorder. Dorder* contains two phases. The first phase is SemiCore*, which is shown by grey color in the figure. The rest of



(a) Time Cost (b) I/Os

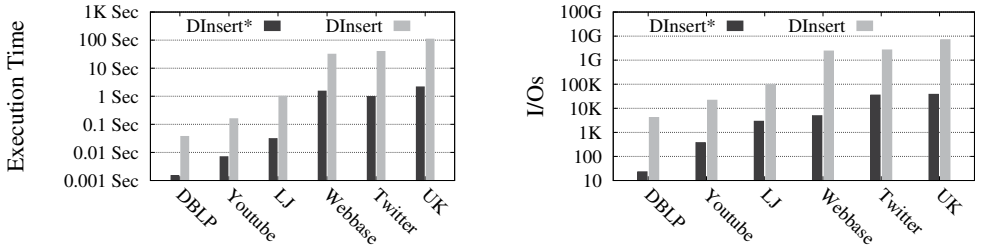
Figure 4.7: Time cost and I/Os of computing degeneracy order

Dorder* is shown by black color. We can see that after core decomposition, we can only spend small costs to compute the degeneracy order for a graph. In Fig. 4.7 (a), the Dorder computes the degeneracy order of the three small datasets DBLP, Youtube, and LiveJournal in 0.1 second, 0.3 second, and 3.8 seconds respectively. For the big datasets, Dorder spends 190 seconds, 256 seconds, and 437 seconds for Webbase, Twitter and UK-2002 respectively. We can see that with the assistance of SemiCore*, Dorder* is more efficient than Dorder.

4.7.4 Degeneracy Order Maintenance

We test the performance of our degeneracy maintenance algorithm DInsert* on three small graphs (DBLP, Youtube, and LiveJournal) and three big graphs (Webbase, Twitter and UK-2002). We do not show the performance for edge removal since the removal of an edge will not trigger the update of the *level-value* for vertices in the Level-Index. For each test, we select 100 edge insertions that will trigger the change of the Level-Index and take the average cost. We compare our algorithm DInsert* with the Dorder* algorithm which works as follows. It first checks whether the graph degeneracy changes using SemiInsert*, if so, it computes the degeneracy order from scratch using Dorder*; otherwise, it invokes line 2-11 of Dorder* (Algorithm 17) to compute the new degeneracy order. Both

DInsert* and DInsert include the algorithm to maintain the core number.



(a) Time Cost (b) I/Os

Figure 4.8: Time cost and I/Os of degeneracy order maintenance

The statistics of degeneracy order maintenance are given in Fig. 4.8. According to Fig. 4.8, on average, DInsert* is 30 times faster than DInsert and spends 100 times less I/Os comparing to DInsert. As an example, on the Webbase dataset, DInsert* spends 1.5 seconds and $4.9K$ I/Os while DInsert cost 31 seconds and $2387K$ I/Os. This demonstrates the high efficiency of our degeneracy order maintenance algorithm.

4.7.5 Scalability Testing

In this experiment, we test the scalability of our core decomposition and core maintenance algorithms. We choose two big graphs Twitter and UK-2002 for testing. We vary number of vertices $|V|$ and number of edges $|E|$ of Twitter and UK-2002 by randomly sampling vertices and edges respectively from 20% to 100%. When sampling vertices, we keep the induced subgraph of the vertices, and when sampling edges, we keep the incident vertices of the edges. Here, we only report the processing time. The memory usage is linear to the number of vertices, and the curves for I/O cost are similar to that of processing time.

Core Decomposition. Fig. 4.9 (a) and (b) report the processing time of our proposed algorithms for core decomposition when varying $|V|$ in Twitter and

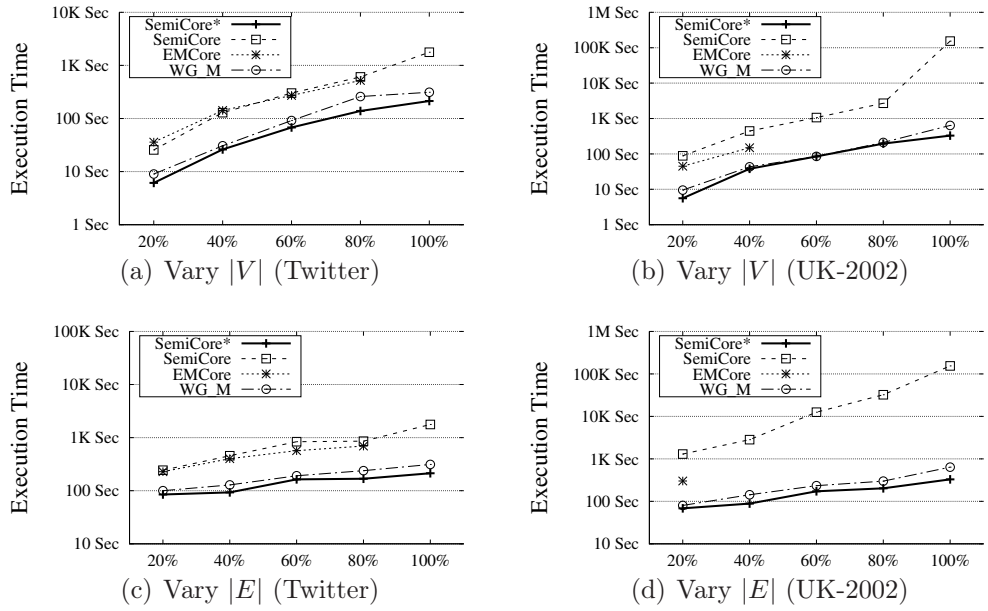


Figure 4.9: Scalability of core decomposition

UK-2002 respectively. When $|V|$ increases, the processing time for all algorithms increases. SemiCore* performs best in all cases and is over an order of magnitude faster than SemiCore in both Twitter and UK-2002. WG_M is the second best. Fig. 4.9 (d) and (d) show the processing time of our core decomposition algorithms when varying $|E|$ in Twitter and UK-2002 respectively. When $|E|$ increases, the processing time for all algorithms increases, and SemiCore* performs best among all three algorithms. When $|E|$ increases, the gap between SemiCore* and SemiCore also increases. For example, in UK-2002, when $|E|$ reaches 100%, SemiCore* is more than two orders of magnitude faster than SemiCore.

Core Maintenance. The scalability testing results for core maintenance are shown in Fig. 4.10. Since there does not exist other work for core maintenance in big graphs, we give the curves for IMInsert and IMDelete as comparisons. As shown in Fig. 4.10 (a) and Fig. 4.10 (b), when increasing $|V|$ from 20% to 100%,

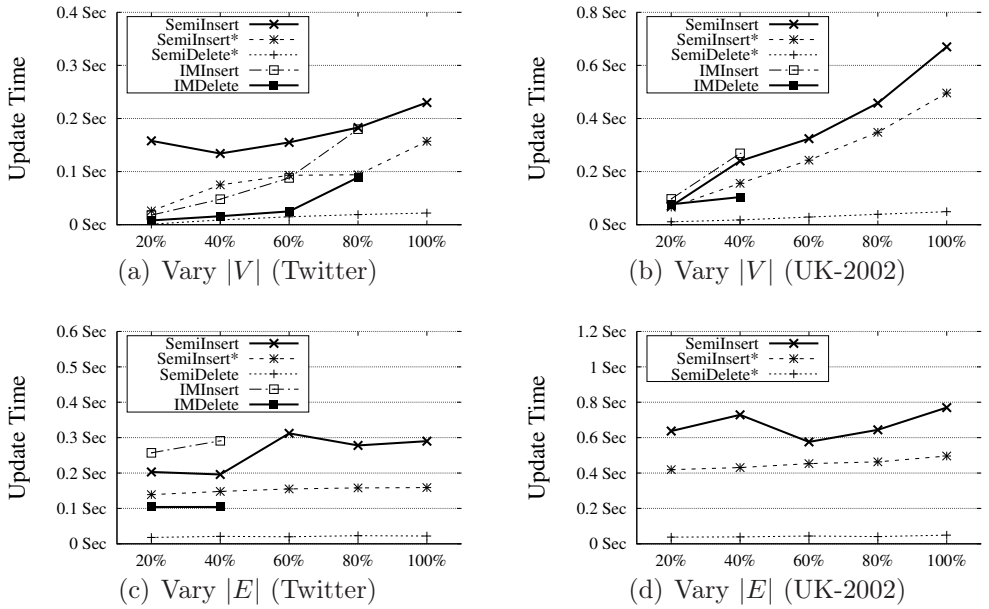
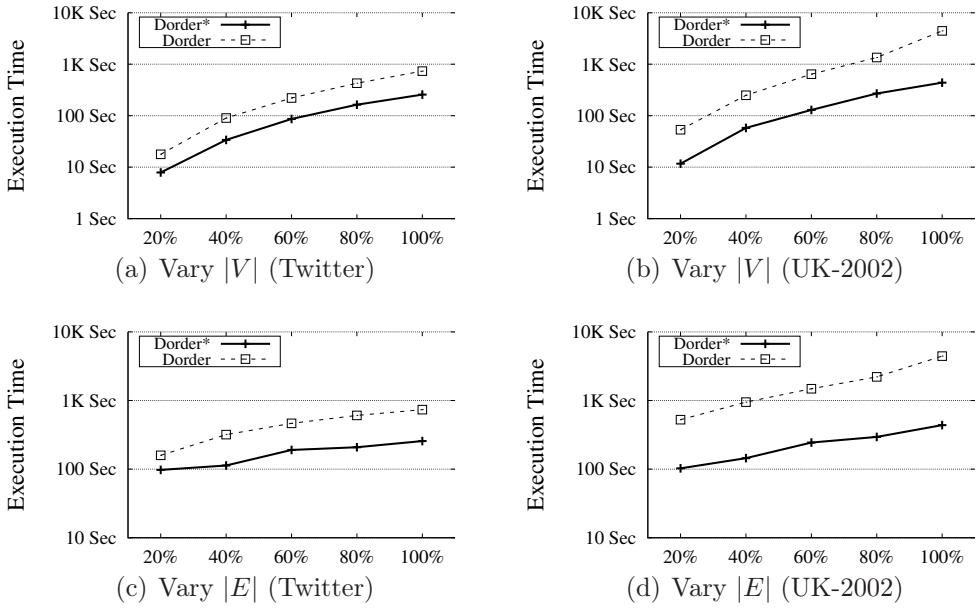


Figure 4.10: Scalability of core maintenance

the processing time for all algorithms increases. SemiDelete* performs best, and SemiInsert* is faster than SemiInsert for all testing cases. The curves of our core maintenance algorithms when varying $|E|$ are shown in Fig. 4.10 (c) and Fig. 4.10 (d) for Twitter and UK-2002 respectively. SemiDelete* and SemiInsert* are very stable when increasing $|E|$ in both Twitter and UK-2002, which shows the high scalability of our core maintenance algorithms. SemiInsert performs worst among all three algorithms. When $|E|$ increases, the performance of SemiInsert is unstable because SemiInsert needs to locate a connected component whose size can be very large in some cases.

Degeneracy Order Computation. Fig. 4.11 (a) and (b) report the processing time of our proposed algorithms for degeneracy ordering when varying $|V|$ in Twitter and UK-2002 respectively. Fig. 4.11 (c) and (d) show the statistics when varying $|E|$. When $|V|$ or $|E|$ increases, the processing time for all algorithms



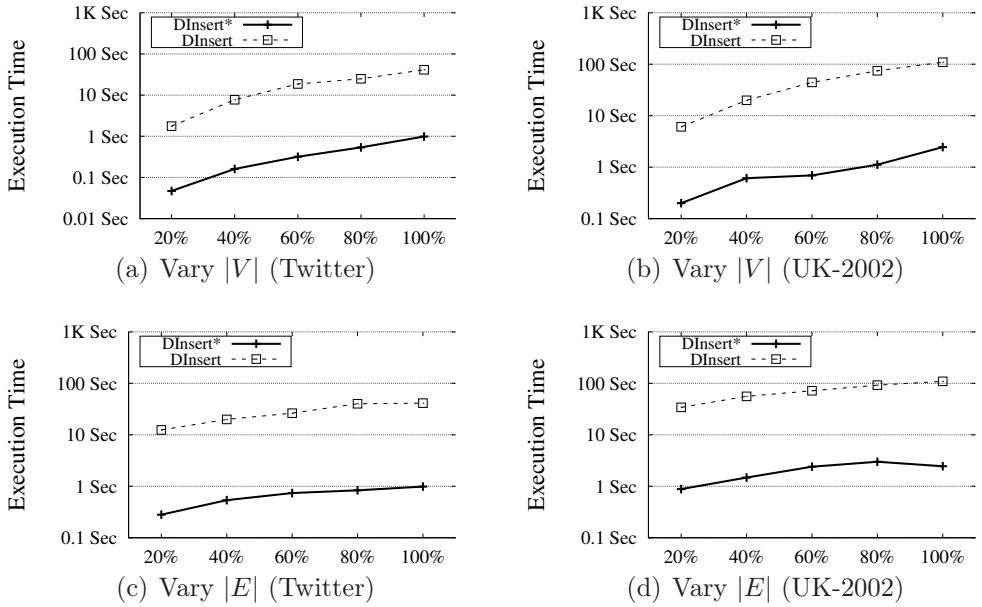
(a) Vary $|V|$ (Twitter) (b) Vary $|V|$ (UK-2002)

(c) Vary $|E|$ (Twitter) (d) Vary $|E|$ (UK-2002)

Figure 4.11: Scalability of degeneracy ordering

increases. Dorder* performs better than Dorder in all cases in both Twitter and UK-2002. Especially, the gap between Dorder* and Dorder increases when $|E|$ increases. When $|E|$ reaches 100% in UK-2002, Dorder* is almost ten times faster than Dorder.

Degeneracy Order Maintenance. Fig. 4.12 (a) and (b) report the processing time of our proposed algorithms for degeneracy order maintenance when varying $|V|$ in Twitter and UK-2002 respectively. Fig. 4.12 (c) and (d) show the statistics when varying $|E|$. When $|V|$ or $|E|$ increases, the processing time for all algorithms increases. DInsert* performs better than DInsert in all cases in both Twitter and UK-2002.



(a) Vary $|V|$ (Twitter) (b) Vary $|V|$ (UK-2002)

(c) Vary $|E|$ (Twitter) (d) Vary $|E|$ (UK-2002)

Figure 4.12: Scalability of degeneracy order maintenance

4.8 Chapter Summary

In this chapter, we study I/O efficient core decomposition on web-scale graphs. The existing solution does not scale to big graphs because it may load most of the graph in memory. Therefore, we follow a semi-external model, which can bound the memory size well. We propose an I/O efficient semi-external algorithm for core decomposition, and explore two optimization strategies to further reduce the I/O and CPU time. We further propose semi-external algorithms and optimization techniques to handle graph updates. We show that our core decomposition and core maintenance algorithms can be respectively used to efficiently compute the degeneracy order in a graph and maintain the order incrementally when the graph dynamically updates. We conduct extensive experiments on 12 real graphs, one of which contains 978.5 million vertices and 42.6 billion edges, to demonstrate the efficiency of our proposed algorithms.

Chapter 5

CONNECTIVITY-CONSTRAINED COMMUNITY DETECTION

5.1 Overview

In this section, we introduce our proposed solution for computing all k-vertex connected components in a given graph. The full version of this work can be found in [84] and the rest of this chapter is organized as follows. Section 5.2 formally defines the problem. Section 5.3 introduces an existing framework to compute all k-VCCs in a given graph. Section 5.4 gives a basic implementation of the framework and analyzes the time complexity of the algorithm. Section 5.5 introduces several strategies to speed up the algorithm. Section 5.6 evaluates the model and algorithms using extensive experiments. Section 5.7 summarizes the chapter.

5.2 Preliminary

We give some basic definitions as follows before stating the problem.

Definition 15. (Vertex Connectivity) *The vertex connectivity of a graph G, denoted by $\kappa(G)$, is defined as the minimum number of vertices whose removal results in either a disconnected graph or a trivial graph (a single-vertex graph).*

Definition 16. (k-Vertex Connected) *A graph G is k-vertex connected if: 1) $|V(G)| > k$; and 2) remaining graph is still connected after removing any $(k-1)$ vertices. That is, $\kappa(G) \geq k$.*

We use the term *k-connected* for short when the context is clear. It is easy to see that any nontrivial connected graph is at least 1-connected. Based on Definiton 16, we define the *k-Vertex Connected Component* (k-VCC) as follows.

Definition 17. (k-Vertex Connected Component) *Given a graph G, a subgraph g is a k-vertex connected component (k-VCC) of G if: 1) g is k-vertex connected; and 2) g is maximal. That is, $\nexists g' \subseteq G$, such that $\kappa(g') \geq k$, $g \subseteq g'$.*

Problem Definition. Given a graph G and an integer k, we denote the set of all k-VCCs of G as $VCC_k(G)$. We study the problem of enumerating all k-VCCs in G, i.e, to compute $VCC_k(G)$.

Example 14. *For the graph G in Fig. 1.2, there are four 4-VCCs: G_1, G_2, G_3 and G_4. We cannot disconnect each of them by removing any 3 or fewer vertices. Subgraph $G_1 \cup G_2$ is not a 4-VCC because it will be disconnected after removing a and b.*

5.3 Algorithm Framework

To compute all k-VCCs in a given graph G, [48] proposes a cut-based framework, which is the state-of-the-art solution for this problem. For the ease of

presentation, we named it by KVCC-ENUM. Before introducing the details of KVCC-ENUM, we define the *vertex cut*.

Definition 18. (Vertex Cut) *Given a connected graph G, a vertex subset $\mathcal{S} \subset V$ is a vertex cut if the removal of $\mathcal{S}$ from G results in a disconnected graph.*

From Definiton 18, we know that the vertex cut may not be unique for a given graph G, and the vertex connectivity is the cardinality of the minimum vertex cut. For a complete graph, there is no vertex cut since any two vertices are adjacent. The size of a vertex cut is the number of vertices in the cut. In the rest of chapter, we use the term cut for short to represent vertex cut when the context is clear.

The Framework. Given a graph G, the general idea of the framework KVCC-ENUM [48] is given as follows. If G is k-connected, G itself is a k-VCC. Otherwise, there must exist a qualified cut $\mathcal{S}$ whose size is less than k. In this case, we compute such vertex cut and partition the graph G into overlapped subgraphs based on the cut. The partition procedure is repeated until each remaining subgraph is a k-VCC. Given that a k-VCC must be a k-core (a graph with minimum degree no smaller than k [9]) [88], all k-cores are computed in advance to reduce the size of the graph.

We arrange the pseudocode of KVCC-ENUM in Algorithm 20. In line 2, it computes k-core by iteratively removing the vertices whose degree is less than k and terminates once no such vertex exists. Then it identifies connected components of the input graph G. For each connected component G_i (line 4), KVCC-ENUM first computes a cut of G_i by invoking the subroutine GLOBAL-CUT (line 5). Here, we only need to find a cut with fewer than k vertices instead of a minimum cut. The detailed implementation of GLOBAL-CUT will be introduced later. If there is no such cut, it means G_i is k-connected and we add it to the result list $VCC_k(G)$ (lines 6-7). Otherwise, the graph is partitioned into over-

Algorithm 20 KVCC-ENUM(G, k)

Input: a graph G and an integer k;
Output: all k-vertex connected components;
1: $VCC_k(G) \leftarrow \emptyset$;
2: **while** $\exists u : deg(u) < k$ **do** remove u and incident edges;
3: identify connected components $\mathcal{G} = \{G_1, G_2, ..., G_t\}$ in G;
4: **for each** connected component $G_i \in \mathcal{G}$ **do**
5: $\quad \mathcal{S} \leftarrow$ GLOBAL-CUT(G_i, k);
6: $\quad$ **if** $\mathcal{S} = \emptyset$ **then**
7: $\quad\quad VCC_k(G) \leftarrow VCC_k(G) \cup \{G_i\}$;
8: $\quad$ **else**
9: $\quad\quad \mathcal{G}_i \leftarrow$ OVERLAP-PARTITION$(G_i, \mathcal{S})$;
10: $\quad\quad$ **for each** $G_i^j \in \mathcal{G}_i$ **do**
11: $\quad\quad\quad VCC_k(G) \leftarrow VCC_k(G) \cup$ KVCC-ENUM(G_i^j, k);
12: **return** $VCC_k(G)$;
13: **Procedure** OVERLAP-PARTITION(Graph G', Vertex Cut $\mathcal{S}$)
14: $\mathcal{G} \leftarrow \emptyset$;
15: remove vertices in $\mathcal{S}$ and their adjacent edges from G';
16: **for each** connected component G_i' of G' **do**
17: $\quad \mathcal{G} \leftarrow \mathcal{G} \cup \{G'[V(G_i') \cup \mathcal{S}]\}$;
18: **return** $\mathcal{G}$;

lapped subgraphs using the cut $\mathcal{S}$ by invoking OVERLAP-PARTITION (line 9). KVCC-ENUM recursively cuts each of other subgraphs using the same procedure KVCC-ENUM (line 11) until all remaining subgraphs are k-VCCs. Next, we introduce the subroutine OVERLAP_PARTITION, which partitions the graph into overlapped subgraphs by cut $\mathcal{S}$.

Overlapped Graph Partition. To partition a graph G into overlapped subgraphs using a cut $\mathcal{S}$, we cannot simply remove all vertices in $\mathcal{S}$, since such vertices may be the overlapped vertices of two or more k-VCCs. Subroutine OVERLAP-PARTITION is shown in lines 13-18 of Algorithm 20. We first remove the vertices in $\mathcal{S}$ along with their adjacent edges from G'. G' will become disconnected after removing $\mathcal{S}$, since $\mathcal{S}$ is a vertex cut of G'. We can simply add the cut $\mathcal{S}$ into each connected component G_i' of G' and return induced subgraph

$G'[V(G'_i) \cup \mathcal{S}]$ as the partitioned subgraph (lines 17-18). Partitioned subgraphs overlap each other since the cut $\mathcal{S}$ is duplicated in these subgraphs. Below, we use an example to illustrate the partition process.

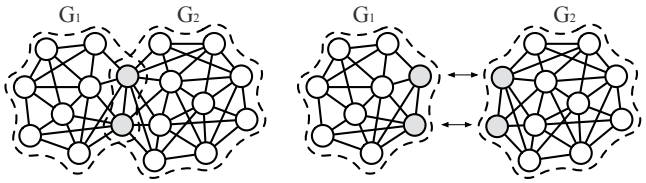


Figure 5.1: An example of overlapped graph partition.

Example 15. *We consider a graph G on the left of Fig. 5.1. Given $k = 3$, we can find a vertex cut in which all vertices are marked by gray. These vertices belong to both 3-VCCs, G_1 and G_2. Thus we can partition G by duplicating the induced subgraph of cut $\mathcal{S}$. As shown on the right of Fig. 5.1, we obtain two 3-VCCs, G_1 and G_2, by duplicating the two cut vertices and their inner edges.*

5.4 Basic Solution

In Algorithm 20, a crucial part is computing a vertex cut of G by invoking GLOBAL-CUT. In this section, we introduce an existing implementation for GLOBAL-CUT [48, 29], and then we explore optimization strategies to accelerate the computation of the vertex cut in Section 5.5.

5.4.1 Find Vertex Cut

We first give some necessary definitions before introducing the basic implementation of GLOBAL-CUT.

Definition 19. (Minimum u-v Cut) *A vertex cut $\mathcal{S}$ is a u-v cut if u and v are in disjoint subsets after removing $\mathcal{S}$, and it is a minimum u-v cut if its size is no larger than that of other u-v cuts.*

Definition 20. (Local Connectivity) *Given a graph G, the local connectivity of two vertices u and v, denoted by $\kappa(u, v, G)$, is defined as the size of the minimum u-v cut. $\kappa(u, v, G) = +\infty$ if no such cut exists.*

Based on Definiton 20, we define two local k connectivity relations as follows:

- $u \equiv_G^k v$: The local connectivity between u and v is not less than k in graph G, i.e., $\kappa(u, v, G) \geq k$.
- $u \not\equiv_G^k v$: The local connectivity between u and v is less than k in graph G, i.e., $\kappa(u, v, G) < k$.

We omit the suffix G when the context is clear. Once $u \equiv^k v$, we say u and v is k-local connected. Obviously, $u \equiv^k v$ is equivalent to $v \equiv^k u$ and following lemma holds.

Lemma 14. *$u \equiv^k v$ if $(u, v) \in E$.*

The GLOBAL-CUT Algorithm. We follow the idea in [48, 29] to implement GLOBAL-CUT. The pseudocode is given in Algorithm 21. Given a graph G, we assume that G contains a vertex cut $\mathcal{S}$ such that $|\mathcal{S}| < k$. Consider an arbitrary source vertex u. There are only two cases: (i) $u \notin \mathcal{S}$ and (ii) $u \in \mathcal{S}$. The general idea of algorithm GLOBAL-CUT is considering these two cases. In the first phase (lines 4-6), we select a vertex u and test the local connectivity between u and all other vertices v in G by invoking LOC-CUT. We have either (a) $u \in \mathcal{S}$ or (b) G is k-connected if each local connectivity is not less than k. In the second phase (lines 7-10), we consider the case $u \in \mathcal{S}$ and test the local connectivity between any two neighbors of u based on Lemma 15.

Algorithm 21 GLOBAL-CUT(G, k)

Input: a graph G and an integer k;
Output: a vertex cut with fewer than k vertices;

```
1: compute a sparse certification SC of G;
2: select a source vertex u with minimum degree;
3: construct the directed flow graph SC' of SC;
4: for each v ∈ V do
5:    S ← LOC-CUT(u, v, SC', SC);
6:    if S ≠ ∅ then return S;
7: for each v_a ∈ N(u) do
8:    for each v_b ∈ N(u) do
9:       S ← LOC-CUT(v_a, v_b, SC', SC);
10:      if S ≠ ∅ then return S;
11: return ∅;
12: Procedure LOC-CUT(u, v, G', G)
13: if v ∈ N(u) or v = u then return ∅;
14: λ ← calculate the maximum flow from u to v in G';
15: if λ ≥ k then return ∅;
16: compute the minimum edge cut in G';
17: return the corresponding vertex cut in G;
```

Lemma 15. *Given a non-k-vertex connected graph G and a vertex $u \in \mathcal{S}$ where $\mathcal{S}$ is a vertex cut and $|\mathcal{S}| < k$, there exist $v, v' \in N(u)$ such that $v \not\equiv^k v'$. [29]*

To test the connectivity of two vertices, we need to transform the original graph into a directed flow graph with $2n$ vertices and $n + 2m$ edges and the capacities of all edges are 1 (line 3). The local connectivity between u and v is equal to the max-flow between them on directed flow graph. LOC-CUT returns the corresponding vertex cut if the max-flow is less than k (lines 15-17). More details about directed flow graph can be found in [29].

Sparse Certificate. Based on the original version of GLOBAL-CUT in [48], we adopt an optimization in Algorithm 21; that is computing a sparse certificate [21] of the original graph (line 1). We introduce the definition of sparse certificate as follows.

Definition 21. (CERTIFICATE) *A certificate for the k-vertex connectivity of G*

is a subset E' of E such that the subgraph (V, E') is k-vertex connected if and only if G is k-vertex connected. [21]

Definition 22. (Sparse Certificate) *A certificate for k-vertex connectivity of G is called sparse if it has $O(k \cdot n)$ edges. [21]*

From the definitions, we can see that a sparse certificate is equivalent to the original graph w.r.t k-vertex connectivity. Meanwhile, it can bound the edge size. We will show that the sparse certificate can not only be used to reduce the graph size, but also used to further reduce local connectivity testings in Section 5.5. The sparse certificate is computed according to the following theorem.

Theorem 15. *Let $G(V, E)$ be an undirected graph and let n denote the number of vertices. Let k be a positive integer. For $i = 1, 2, ..., k$, let E_i be the edge set of a scan first search forest F_i in the graph $G_{i-1} = (V, E - (E_1 \cup E_2 \cup ... \cup E_{i-1}))$. Then $E_1 \cup E_2 \cup ... \cup E_k$ is a certificate for the k-vertex connectivity of G, and this certificate has at most $k \times (n - 1)$ edges [21].*

Based on Theorem 15, we can construct a sparse certificate of G using scan first search k times, each of which creates a scan first search forest F_i. More details about scan first search can be found in [84, 21]. Note that breath first search is a special case of scan first search and thus can be used here. An example of constructing sparse certificate is given as follows.

Example 16. *Fig. 5.2 presents construction of a sparse certificate for graph G. Let $k = 3$. For $i \in \{1, 2, 3\}$, F_i denotes the scan first search forest obtained from G_{i-1}. G_i is obtained by removing the edges in F_i from G_{i-1}. G_0 is the input graph G. The obtained sparse certificate SC is shown on the right side of G with $SC = F_1 \cup F_2 \cup F_3$. All removed edges are shown in G_3.*

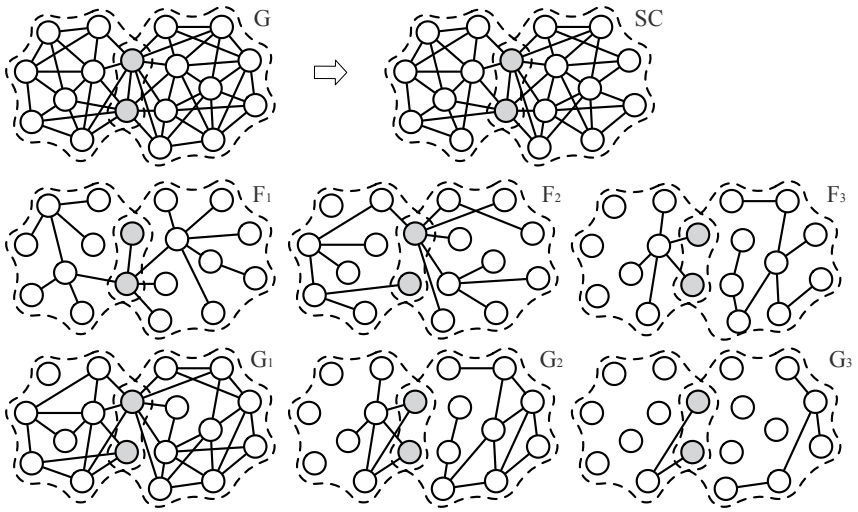


Figure 5.2: The sparse certificate of given graph G with $k = 3$

5.4.2 Algorithm Analysis

Given that there does not exist any theoretical analysis for the running time of the overall framework KVCC-ENUM in [48], we prove that KVCC-ENUM terminates in polynomial time in this section. In the directed flow graph, all edge capacities are equal to 1 and every vertex either has a single edge emanating from it or has a single edge entering it. For this kind of graph, the time complexity for computing the maximum flow is $O(n^{0.5}m)$ [31]. Note that we do not need to calculate the exact flow value in the algorithm. Once the flow value reaches k, we know that local connectivity between any two given vertices is at least k and we can terminate the maximum flow algorithm. The time complexity for the flow computation is $O(\min(n^{0.5}, k) \cdot m)$. Given a flow value and corresponding residual network, we can perform a depth first search to find the cut. It costs $O(m + n)$ time. As a result, we have the following lemma:

Lemma 16. *The CPU time complexity of algorithm* LOC-CUT *is* $O(\min(n^{0.5}, k) \cdot m)$.

Next we discuss the time complexity of GLOBAL-CUT. The construction of both sparse certificate and directed flow graph costs $O(m+n)$ CPU time. Let δ denote the minimum degree in the input graph. We can easily get following lemma.

Lemma 17. GLOBAL-CUT *invokes* LOC-CUT $O(n+\delta^2)$ *times in the worst case.*

Below we discuss the time complexity of the entire algorithm KVCC-ENUM. KVCC-ENUM iteratively removes vertices with degree less than k (line 2). This costs $O(m+n)$ time. Identifying all connected components (line 3) can be performed by adopting a depth first search in $O(m+n)$ time. To study the number of times that GLOBAL-CUT is invoked, we first give the following lemmas.

Lemma 18. *For each connected component C obtained by overlapped partition in Algorithm 20, $|V(C)| \geq k+1$.*

Proof. Let $\mathcal{S}$ denote a vertex cut in an overlapped partition. C is one of the connected components obtained in this partition. Let H denote the vertex set of all vertices in $V(C)$ but not in $\mathcal{S}$, i.e., $H = \{u | u \in V(C), u \notin \mathcal{S}\}$. We have $H \neq \emptyset$. Note that each vertex in the graph has a degree at least k in G (line 5 in Algorithm 20). There exist at least k neighbors for each vertex u in H and therefore for each neighbor v of u we have $v \in C$ according to Lemma 14. Thus, we have $|V(C)| \geq k+1$. □

Lemma 19. *The total number of overlapped partitions during the algorithm* KVCC-ENUM *is not larger than* $\frac{n-k-1}{2}$.

Proof. Suppose that λ is the total number of overlapped partitions during the whole algorithm KVCC-ENUM. This generates at least $\lambda+1$ connected components. We know from Lemma 18 that each connected component contains at least $k+1$ vertices. Thus, we have at least $(\lambda+1)(k+1)$ vertices in total. On

the other hand, we increase at most $k-1$ vertices in each subgraph obtained by an overlapped partition. Thus, at most $\lambda(k-1)$ vertices are added. We have $(\lambda+1)(k+1) \leq n+\lambda(k-1)$. Rearranging the formula, we obtain $\lambda \leq \frac{n-k-1}{2}$. □

We then prove the upper bound for number of k-VCCs.

Theorem 16. *Given a graph G and an integer k, the number of k-VCCs is bounded by $\frac{n}{2}$, i.e., $|VCC_k(G)| < \frac{|V(G)|}{2}$.*

Proof. Similar to the proof of Lemma 19, let λ be the times of overlapped partitions in the whole algorithm KVCC-ENUM. At most $\lambda(k-1)$ vertices are increased. Let σ be the number of connected components obtained in all partitions. We have $\sigma > \lambda$. Each connected component contains at least $k+1$ vertices according to Lemma 18. Note that each connected component is either a k-VCC or a graph that does not contain any k-VCC. Otherwise, the connected component will be further partitioned. Let x be the number of k-VCCs and y be the number of connected components that do not contain any k-VCC, i.e., $x+y=\sigma$. We know that a k-VCC contains at least $k+1$ vertices. Thus there are at least $x(k+1)+y(k+1)$ vertices after finishing all partitions. We obtain $x(k+1)+y(k+1) \leq n+\lambda(k-1)$. Since $\lambda < \sigma$ and $\sigma = x+y$, we rearrange the formula as $x(k+1)+y(k+1) < n+x(k-1)+y(k-1)$. Therefore, we have $x < \frac{n}{2}$. □

Theorem 17. *The total time complexity of* KVCC-ENUM *is* $O(\min(n^{1/2}, k) \cdot m \cdot (n+\delta^2) \cdot n)$.

Proof. The total time complexity of KVCC-ENUM is dependent on the number of times GLOBAL-CUT is invoked. Suppose GLOBAL-CUT is invoked p times during the whole KVCC-ENUM algorithm, the number of overlapped partitions during the whole KVCC-ENUM algorithm is p_1 and the total number of k-VCCs is p_2. It

is easy to see that $p = p_1 + p_2$. From Lemma 19, we know that $p_1 \leq \frac{n-k-1}{2} < \frac{n}{2}$. From Theorem 16, we know that $p_2 < \frac{n}{2}$. Therefore, we have $p = p_1 + p_2 < n$. According to Lemma 16 and Lemma 17, the theorem holds. □

Discussion. Theorem 17 shows that all k-VCCs can be enumerated in polynomial time. Although the time complexity is still high, it performs much better in practice. Note that the time complexity is the product of three parts:

- The first part $O(\min(n^{1/2}, k) \cdot m)$ is the time complexity for LOC-CUT to test whether there exists a vertex cut of size smaller than k. In practice, the graph to be tested is much smaller than the original graph G since (1) The graph to be tested has been pruned using the k-core technique and sparse certification technique. (2) Due to the graph partition scheme, the input graph is partitioned into many smaller graphs.

- The second part $O(n + \delta^2)$ is the number of times such that LOC-CUT (local connectivity testing) is invoked by the algorithm GLOBAL-CUT. We will discuss how to significantly reduce the number of local connectivity testings in Section 5.5.

- The third part $O(n)$ is the number of times GLOBAL-CUT is invoked. In practice, the number can be significantly reduced since the number of k-VCCs is usually much smaller than $\frac{n}{2}$.

In the next section, we will explore several search reduction techniques to speed up the algorithm.

5.5 Search Reduction

In the previous section, we introduce a basic solution for k-VCC enumeration. Recall that in the worst case, we need to test local connectivity between the

source vertex u and all other vertices in G in GLOBAL-CUT, and we also need to test local connectivity for every pair of neighbors of u. For each pair of vertices, we need to compute the maximum flow in the directed flow graph. Therefore, the key to improving the algorithm is to reduce the number of local connectivity testings (LOC-CUT). In this section, we propose several techniques to avoid unnecessary testings. We can avoid testing local connectivity of a vertex pair (u, v) if we can guarantee that $u \equiv^k v$. We call such operation a sweep operation. Below, we introduce two ways to efficiently prune unnecessary testings, namely neighbor sweep and group sweep, in Section 5.5.1 and Section 5.5.2 respectively.

5.5.1 Neighbor Sweep

In this section, we propose a neighbor sweep strategy to prune unnecessary local connectivity testings (LOC-CUT) in the first phase of GLOBAL-CUT. Generally speaking, given a source vertex u, for any vertex v, we aim to skip testing the local connectivity of (u, v) according to the information of the neighbors of v. Below, we explore two neighbor sweep strategies, namely neighbor sweep using side-vertex and neighbor sweep using vertex deposit.

Neighbor Sweep using Side-Vertex

We first define *side-vertex* as follows.

Definition 23. (SIDE-VERTEX) *Given a graph G and an integer k, a vertex u is called a side-vertex if there does not exist a vertex cut $\mathcal{S}$ such that $|\mathcal{S}| < k$ and $u \in \mathcal{S}$.*

Based on Definiton 23, we give the following lemma to show the transitive property regarding the local k connectivity relation $\equiv^k$.

Lemma 20. *Given a graph G and an integer k, suppose $a \equiv^k b$ and $b \equiv^k c$, we have $a \equiv^k c$ if b is a side-vertex.*

Proof. We prove it by contradiction. Assume that b is a side-vertex and $a \not\equiv^k c$. There exists a vertex cut with $k-1$ or fewer vertices between a and c. b is not in any such cut since it is a side-vertex. Then we have either $b \not\equiv^k a$ or $b \not\equiv^k c$. This contradicts the precondition that $a \equiv^k b$ and $b \equiv^k c$. □

A wise way to use the transitive property of the local connectivity relation in Lemma 20 can largely reduce the number of unnecessary testings. Consider a selected source vertex u in algorithm GLOBAL-CUT. We assume that LOC-CUT (line 5) returns $\emptyset$ for a vertex v, i.e., $u \equiv^k v$. We know from Lemma 20 that the vertex pair (u, w) can be skipped for local connectivity testing if (i) $v \equiv^k w$ and (ii) v is a side-vertex. For condition (i), we can use a simple necessary condition according to Lemma 14, that is, for any vertices v and w, $v \equiv^k w$ if $(v, w) \in E$. In the following, we focus on condition (ii) and look for necessary conditions to efficiently check whether a vertex is a side-vertex.

Side-Vertex Detection. To check whether a vertex is a side-vertex, we can easily obtain the following lemma based on Definiton 23.

Lemma 21. *Given a graph G, a vertex u is a side-vertex if and only if $\forall v, v' \in N(u)$, $v \equiv^k v'$.*

Recall that two vertices are k-local connected if they are neighbors of each other. For the k-local connectivity of non-connected vertices, we give another necessary condition below.

Lemma 22. *Given two vertices u and v, $u \equiv^k v$ if $|N(u) \cap N(v)| \geq k$.*

Combining Lemma 21 and Lemma 22, we derive the following necessary condition to check whether a vertex is a side-vertex.

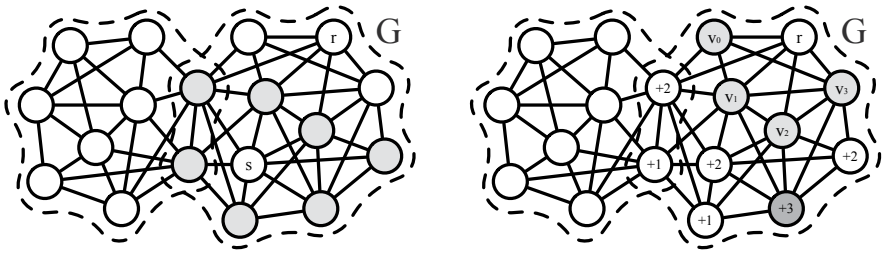
Theorem 18. *A vertex u is a side-vertex if $\forall v, v' \in N(u)$, either $(v, v') \in E$ or $|N(v) \cap N(v')| \geq k$.*

Definition 24. (Strong Side-Vertex) *A vertex u is called a strong side-vertex if it satisfies the conditions in Theorem 18.*

Using strong side-vertex, we define our first neighbor sweep rule.

(Neighbor Sweep Rule 1) *Given a graph G and an integer k, let u be a selected source vertex in algorithm* GLOBAL-CUT *and v be a strong side-vertex in the graph. We can skip the local connectivity testings of all pairs of (u, w) if we have $u \equiv^k v$ and $w \in N(v)$.*

We give an example to demonstrate *neighbor sweep rule 1* below.



(a) A strong side-vertex s (b) Vertex deposit

Figure 5.3: Strong side-vertex and vertex deposit when $k = 3$

Example 17. *Fig. 5.3 (a) presents a strong side-vertex s in graph G while parameter $k = 3$. Assume that r is the source vertex. Any two neighbors of s are either connected by an edge or have at least 3 common neighbors. If first test the local connectivity between r and s and $r \equiv^k s$, we can safely sweep all neighbors of s, which are marked by the gray color in Fig. 5.3 (a).*

Below, we discuss how to efficiently detect all strong side-vertices and maintain strong side-vertices when the graph is partitioned.

Strong Side-Vertex Computation. Following Theorem 18, we can compute all strong side-vertices v in advance and skip all neighbors of v once v is k connected with the source vertex (line 5 in GLOBAL-CUT). We can derive the following lemma.

Lemma 23. *The time complexity of computing all strong side-vertices in graph G is $O(\sum_{w \in V(G)} deg(w)^2)$.*

Proof. To compute all strong side-vertices in a graph G, we first check all 2-hop neighbors v for each vertex u. Since v and u share a common vertex of 1-hop neighbor, we can easily obtain all vertices which have k common neighbors with u. Any vertex w is considered as 1-hop neighbor of other vertices $deg(w)$ times. We use $deg(w)$ steps to obtain 2-hop neighbors of u which share a common vertex w with u. This phase costs $O(\sum_{w \in V(G)} deg(w)^2)$ time.

Now for each given vertex u, we have all vertices v sharing k common neighbors with it. For each vertex w, we check whether any two neighbors of w have k common neighbors. This phase also costs $O(\sum_{w \in V(G)} deg(w)^2)$ time. Consequently, the total time complexity is $O(\sum_{w \in V(G)} deg(w)^2)$. □

After computing all strong side-vertices for the original graph G, we do not need to recompute strong side-vertices in the partitioned graph from scratch. Instead, we can reduce the number of strong side-vertex checks by making use of the already computed strong side-vertices in G. We efficiently detect non-strong side-vertices and strong side-vertices respectively based on Lemma 24 and Lemma 25.

Lemma 24. *Let G be a graph and G_i be one of the graphs obtained by partitioning G using* OVERLAP-PARTITION *in Algorithm 20, a vertex is a strong side-vertex in G if it is a strong side-vertex in G_i.*

Proof. The strong side-vertex u requires at least k common neighbors between any two neighbors of u. The lemma is obvious since G contains all edges and vertices in G_i. □

From Lemma 24, we know that a vertex is not a strong side-vertex in G_i if it is not a strong side-vertex in G. This property allows us checking limited number of vertices in G_i, which is the set of strong side-vertices in G.

Lemma 25. *Let G be a graph, G_i be one of the graphs obtained by partitioning G using* OVERLAP-PARTITION *in Algorithm 20, and $\mathcal{S}$ is a vertex cut of G, for any vertex $v \in V(G_i)$, if v is a strong side-vertex in G and $N(v) \cap \mathcal{S} = \emptyset$, then v is also a strong side-vertex in G_i.*

Proof. The qualification of a strong side-vertex of vertex v requires the information about two-hop neighbors of v. Vertices in $\mathcal{S}$ are duplicated when partitioning the graph. Given a strong side-vertex v in G, if $N(v) \cap \mathcal{S} = \emptyset$, the two-hop neighbors of v are not affected by the partition operation, thus the relationships between the vertices in $N(v)$ are not affected by the partition operation. Therefore, v is still a strong side-vertex in G_i according to Definiton 24. □

With Lemma 24 and Lemma 25, in a graph G_i partitioned from graph G by vertex cut $\mathcal{S}$, we can reduce the scope of strong side-vertex checks from the vertices in the whole graph G_i to the vertices u satisfying following two conditions simultaneously:

- u is a strong side-vertex in G; and
- $N(u) \cap \mathcal{S} \neq \emptyset$.

Neighbor Sweep using Vertex Deposit

Vertex Deposit. The strong side-vertex strategy heavily relies on the number of strong side-vertices. Next, we investigate a new strategy called vertex deposit, to further sweep vertices based on neighbor information. We first give the following lemma:

Lemma 26. *Given a source vertex u in graph G, for any vertex $v \in V(G)$, we have $u \equiv^k v$ if there exist k vertices $w_1, w_2, \ldots, w_k$ such that $u \equiv^k w_i$ and $w_i \in N(v)$ for any $1 \leq i \leq k$.*

Proof. We prove it by contradiction. Assume that $u \not\equiv^k v$. There exists a vertex cut $\mathcal{S}$ with $k-1$ or fewer vertices between u and v. For any $w_i (1 \leq i \leq k)$, we have $w_i \equiv^k v$ since $w_i \in N(v)$ (Lemma 14) and we also have $w_i \equiv^k u$. Since $u \not\equiv^k v$, w_i cannot satisfy both $w_i \equiv^k u$ and $w_i \equiv^k v$ unless $w_i \in \mathcal{S}$. Therefore, we obtain a cut $\mathcal{S}$ with at least k vertices w_1, w_2, ..., w_k. This contradicts $|\mathcal{S}| < k$. □

Based on Lemma 26, given a source vertex u, once we find a vertex v with at least k neighbors w_i satisfying $u \equiv^k w_i$, we obtain $u \equiv^k v$ without testing their local connectivity. To efficiently detect such vertices, we define *deposit* of each vertex as follows.

Definition 25. *(Vertex Deposit) Given a source vertex u, the deposit for each vertex v, denoted by deposit(v), is the number of neighbors w of v such that $u \equiv^k w$.*

According to Definiton 25, suppose u is the source vertex and for each vertex v, *deposit*(v) is a dynamic value depending on the number of processed vertex pairs. The vertex deposit for each vertex is initialized by 0. Once we know $w \equiv^k u$ for a vertex w, we can increase *deposit*(v) for each $v \in N(w)$ by 1. We obtain the following theorem according to Lemma 26.

Theorem 19. *Given a source vertex u, for any vertex v, we have $u \equiv^k v$ if $deposit(v) \geq k$.*

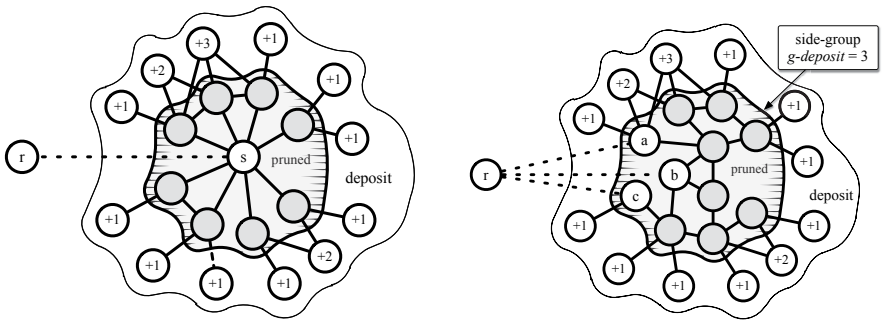
Proof. We proof it by contradiction. Assume that $u \not\equiv^k v$. There exists a vertex cut $\mathcal{S}$ with $k-1$ or less vertices between u and v. Let w_i denote the vertex such that $w_i \in N(v), u \equiv^k w_i$. Here $i \in \{0, 1, ..., k-1\}$ since $deposit(v) = k$. It is obvious that w_i cannot satisfy both $w_i \equiv^k u$ and $w_i \equiv^k v$ unless $w_i \in \mathcal{S}$. Thus, we have at least k vertices in cut $\mathcal{S}$. This contradicts that $|\mathcal{S}| < k$. □

Based on Theorem 19, we derive our second neighbor sweep rule.

(Neighbor Sweep Rule 2) *Given a selected source vertex u, we can skip the local connectivity testing of pair (u, v) if $deposit(v) \geq k$.*

Example 18. *Fig. 5.3 (b) gives an example of vertex deposit strategy. Given the graph G and parameter $k = 3$, let vertex r be the selected source vertex. We assume v_0, v_1, v_2 and v_3 are tested vertices. All these vertices are local k-connected with vertex r, i.e., $r \equiv^k v_i, i \in 0, 1, 2, 3$, since v_0, v_1, v_2 and v_3 are neighbors of r. We deposit once for the neighbors of each tested vertex. The deposit value for each influenced vertices is given in the figure. We mark the vertices with deposit no less than 3 by dark gray. The local connectivity testing between r and such a vertex can be skipped.*

To increase the deposit of a vertex v, we only need any neighbor of v is local k-connected with the source vertex u. We can also use vertex deposit strategy when processing strong side-vertices. Given a source vertex u and a strong side-vertex v, we sweep all $w \in N(v)$ if $u \equiv^k v$. Then we increase the deposit for each non-swept vertex $w' \in N(w)$. In other words, for a strong side-vertex, we can possibly sweep its 2-hop neighbors by combining the two neighbor sweep strategies. An example is given below.



(a) 2-hop deposit (b) group sweep and deposit

Figure 5.4: Increasing deposit with neighbor and group sweep

Example 19. *Fig. 5.4 (a) shows the process for a strong side-vertex s. Given a source vertex r, assume s is a strong side-vertex and $r \equiv^k s$. All neighbors of s are swept and all 2-hop neighbors of s increase their deposits accordingly. The increased value of the deposit for each vertex is shown in the figure. It depends on the number of connected vertices that are swept.*

5.5.2 Group Sweep

The neighbor sweep strategy only prune unnecessary local connectivity testings in the first phase of GLOBAL-CUT by using the neighborhood information. In this subsection, we introduce a new pruning strategy namely group sweep, which can prune unnecessary local connectivity testings in a batch manner. In group sweep, we do not limit the swept vertices to the neighbors of certain vertices. Specifically, we aim to partition vertices into vertex groups and sweep a whole group when it satisfies certain conditions. In addition, our group sweep strategy can also be applied to reduce unnecessary local connectivity testings in both phases of GLOBAL-CUT.

First, we define a new relation regarding a vertex u and a set of vertices C

as follows.

$$u \equiv^k C\text{: For all vertices } v \in C, u \equiv^k v.$$

Given a source vertex u and a side-vertex v, we assume $u \equiv^k v$. According to the transitive relation in Lemma 20, we skip testing the pairs of vertices u and w if $w \equiv^k v$. In our neighbor sweep strategy, we select all neighbors of v as such vertices w, i.e., $u \equiv^k N(v)$. To sweep more vertices each time, we define the side-group.

Definition 26. (SIDE-GROUP) *Given a graph G and an integer k, a vertex set $\mathcal{CC}$ in G is a side-group if $\forall u, v \in C, u \equiv^k v$.*

Note that it is possible that a side-group contains vertices which belongs to a cut $\mathcal{S}$ with $|\mathcal{S}| < k$. Next, we introduce how to construct the side-groups, and then discuss our group sweep rules.

Side-Group Construction. Section 5.4.1 introduces sparse certificate to bound the graph size. Let F_i and G_i be the notations defined in Theorem 15. Assume G is not k-connected and there exists a vertex cut $\mathcal{S}$ with $|\mathcal{S}| < k$. We introduce the following lemma.

Lemma 27. *F_k does not contain a simple tree path P_k whose two end points are in different connected components of $G - \mathcal{S}$. [21]*

Based on Lemma 27, we can obtain the following theorem.

Theorem 20. *Let $\mathcal{CC}$ denote the vertex set of any connected component in F_k. $\mathcal{CC}$ is a side-group.*

Proof. Assume that $u \not\equiv^k v$ in $\mathcal{CC}$. All simple paths from u to v will cross the vertex cut $\mathcal{S}$. This contradicts Lemma 27. □

Example 20. *Review the construction of a sparse certificate in Fig. 5.2. Given* $k = 3$*, two connected components with more than one vertex are obtained in* F_3*. The number of vertices in the two connected components are* 6 *and* 9 *respectively. Each of them is a side-group and any two vertices in the same connected component is local* 3*-connected. Note that the connected component with* 6 *vertices contains two vertices in the vertex cut as marked by gray.*

We denote all side-groups as $\mathcal{CS} = \{\mathcal{CC}_1, \mathcal{CC}_2, \ldots, \mathcal{CC}_t\}$. According to Theorem 20, $\mathcal{CS}$ can be easily computed as a by-product of the sparse certificate. With $\mathcal{CS}$, according to the transitive relation in Lemma 20, we can easily obtain the following pruning rule.

(Group Sweep Rule 1) *Let* u *be the source vertex in the algorithm* GLOBAL-CUT*, given a side-group* $\mathcal{CC}$*, if there exists a strong side-vertex* $v \in \mathcal{CC}$ *such that* $u \equiv^k v$*, we can skip the local connectivity testings of vertex pairs* (u, w) *for all* $w \in \mathcal{CC} - \{v\}$*.*

The above group sweep rule relies on the successful detection of a strong side-vertex in a certain side-group. In the following, we further introduce a deposit based scheme to handle the scenario that no strong side-vertex exists in a certain side-group.

Group Deposit. Similar with the vertex deposit strategy, the group deposit strategy aims to deposit the values in a group level. To show our group deposit scheme, we first introduce the following lemma.

Lemma 28. *Given a source vertex* u*, an integer* k*, and a side-group* $\mathcal{CC}$*, we have* $u \equiv^k \mathcal{CC}$ *if* $|\{v | v \in \mathcal{C}, u \equiv^k v\}| \geq k$*.*

Proof. We prove it by contradiction. Assume that there exists a vertex w in $\mathcal{CC}$ such that $u \not\equiv^k w$. A vertex cut $\mathcal{S}$ exists with $|\mathcal{S}| < k$. Let $v_0, v_1, ..., v_{k-1}$ be the k vertices in $\mathcal{CC}$ such that $u \equiv^k v_i, 0 \leq i \leq k-1$. We have $w \equiv^k v_i$ based on the

definition of a side-group. Each v_i must belong to $\mathcal{S}$ since $u \not\equiv^k w$. As a result, the size of $\mathcal{S}$ is at least k. This contradicts $|\mathcal{S}| < k$. □

Based on Lemma 28, given a source vertex u, once we find a side-group $\mathcal{CC}$ with at least k vertices v with $u \equiv^k v$, we can get $u \equiv^k \mathcal{CC}$ without testing the local connectivity from u to other vertices in $\mathcal{CC}$. To efficiently detect such side-groups $\mathcal{CC}$, we define the group deposit of a side-group $\mathcal{CC}$ as follows.

Definition 27. *(Group Deposit) The group deposit for each side-group $\mathcal{CC}$, denoted by g-deposit($\mathcal{CC}$), is the number of vertices $v \in \mathcal{CC}$ such that $u \equiv^k v$.*

According to Definiton 27, suppose u is the source vertex, for each side-group $\mathcal{CC} \in \mathcal{CS}$, *g-deposit*($\mathcal{CC}$) is a dynamic value depending on the already processed vertex pairs. The group deposit for each side-group $\mathcal{CC}$ is initialized by 0. Once $v \equiv^k u$ for a certain vertex $v \in \mathcal{CC}$, we can increase *g-deposit*($\mathcal{CC}$) by 1. We obtain the following theorem according to Lemma 28.

Theorem 21. *Given a source vertex u, for any side-group $\mathcal{CC} \in \mathcal{CS}$, we have $u \equiv^k \mathcal{CC}$ if g-deposit($\mathcal{CC}$) $\geq k$.*

Based on Theorem 21, we derive our second group sweep rule.

(Group Sweep Rule 2) *Given a selected source vertex u, we can skip the local connectivity testings between u and vertices in $\mathcal{CC}$ if g-deposit($\mathcal{CC}$) $\geq k$.*

Note that a group sweep operation can further trigger a neighbor sweep operation and vice versa, since both operations result in new local k-connected vertex pairs. We show an example below.

Example 21. *Fig. 5.4 (b) presents an example of group sweep. Suppose $k = 3$ and the gray area is a detected side-group. Given a source vertex r, assume that a, b, c are the tested vertices with $r \equiv^k a, r \equiv^k b$ and $r \equiv^k c$ respectively. According to Theorem 21, we can safely sweep all vertices in the same side-group.*

Also, we apply the vertex deposit strategy for neighbors outside the side-group. The increased value of deposit is shown on each vertex.

Next we show that the side-groups can also be used to prune the local connectivity testings in the second phase of GLOBAL-CUT. Recall that in the second phase of GLOBAL-CUT, given a source vertex u, we need to test the local connectivity of every pair (v_a, v_b) of the neighbors of u. With side-groups, we can easily obtain the following group sweep rule.

(Group Sweep Rule 3) *Let u be the source vertex, and v_a and v_b be two neighbors of u. If v_a and v_b belong to the same side-group, we have $v_a \equiv^k v_b$ and thus we do not need to test the local connectivity of (v_a, v_b) in the second phase of* GLOBAL-CUT.

The detailed implementation of the neighbor sweep and group sweep techniques is given in the following section.

5.5.3 The Overall Algorithm

In this section, we combine our pruning strategies and give the implementation of optimized algorithm GLOBAL-CUT*. The pseudocode is presented in Algorithm 22. We can replace GLOBAL-CUT with GLOBAL-CUT* in KVCC-ENUM to obtain our final algorithm to compute all k-VCCs.

The GLOBAL-CUT* algorithm still follows the similar idea of GLOBAL-CUT that consider a source vertex u, and then compute the vertex cut in two phases based on whether u belongs to the vertex cut $\mathcal{S}$. Given a source vertex u, phase 1 (lines 8-15) considers the case that $u \notin \mathcal{S}$. Phase 2 (lines 16-21) considers the case that $u \in \mathcal{S}$. If in both phase, the vertex cut $\mathcal{S}$ is not found, there is no such a cut and we simply return $\emptyset$ in line 22.

We compute the side-groups $\mathcal{CS}$ while computing the sparse certificate (line 1).

Algorithm 22 GLOBAL-CUT*(G, k)

Input: a graph G and an integer k;
Output: a vertex cut with size smaller than k;

1: compute a sparse certification $\mathcal{SC}$ of G and collect all side-groups as $\mathcal{CS} = \{\mathcal{CC}_1, ..., \mathcal{CC}_t\}$;
2: construct the directed flow graph $\overline{\mathcal{SC}}$ of $\mathcal{SC}$;
3: $\mathcal{SV} \leftarrow$ compute all strong side vertices in $\mathcal{SC}$;
4: **if** $\mathcal{SV} = \emptyset$ **then**
5: select a vertex u with minimum degree;
6: **else**
7: randomly select a vertex u from $\mathcal{SV}$;
8: **for all** $\mathcal{CC}_i$ in $\mathcal{CS}$: $g\text{-}deposit(\mathcal{CC}_i) \leftarrow 0$;
9: **for all** v in V: $deposit(v) \leftarrow 0, pru(v) \leftarrow$ false;
10: SWEEP$(u, pru, deposit, g\text{-}deposit, \mathcal{CS})$;
11: **for each** $v \in V$ in non-ascending order of $dist(u, v, G)$ **do**
12: **if** $pru(v) =$ true **then continue**;
13: $\mathcal{S} \leftarrow$ LOC-CUT$(u, v, \overline{\mathcal{SC}}, \mathcal{SC})$;
14: **if** $\mathcal{S} \neq \emptyset$ **then return** $\mathcal{S}$;
15: SWEEP$(v, pru, deposit, g\text{-}deposit, \mathcal{CS})$;
16: **if** u is not a strong side-vertex **then**
17: **for each** $v_a \in N(u)$ **do**
18: **for each** $v_b \in N(u)$ **do**
19: **if** v_a and v_b are in the same $\mathcal{CC}_i$ **then continue**;
20: $\mathcal{S} \leftarrow$ LOC-CUT$(u, v, \overline{\mathcal{SC}}, \mathcal{SC})$;
21: **if** $\mathcal{S} \neq \emptyset$ **then return** $\mathcal{S}$;
22: **return** $\emptyset$;

Note that here we only consider the side-group whose size is larger than k, since the group can be swept only if at least k vertices in the group are swept according to Theorem 21. Then we compute all strong side-vertices, $\mathcal{SV}$ based on Theorem 18 (line 3). Here, the strong side-vertices are computed based on the method discussed in Section 5.5.1. If $\mathcal{SV}$ is not empty, we can select one inside vertex as source vertex u and do not need to consider the phase 2, because u cannot be in any cut $\mathcal{S}$ with $|\mathcal{S}| < k$ in this case. Otherwise, we still select the source vertex u with the minimum degree (lines 4-7).

In phase 1 (lines 8-15), we initialize the group deposit for each side-group,

which is number of swept vertices in the side-group, to 0 (line 8). Also, we initialize the local deposit for each vertex to 0 and pru for each vertex to *false* (line 9). Here, pru is used to mark whether a vertex can be swept. Since the source vertex u is local k-connected with itself, we first apply the sweeping rules on the source vertex by invoking SWEEP procedure (line 10). Intuitively, a vertex that is close to the source vertex u tends to be in the same k-VCC with u. In other words, a vertex v that is far away from u tends to be separated from u by a vertex cut $\mathcal{S}$. Therefore, we process vertices v in G according to the non-ascending order of $dist(u, v, G)$ (line 11). We aim to find the vertex cut by processing as few vertices as possible. For each vertex v to be processed in phase 1, we skip it if $pru(v)$ is true (line 12). Otherwise, we test the local connectivity of u and v using LOC-CUT (line 13). If there is a cut $\mathcal{S}$ with size smaller than k, we simply return $\mathcal{S}$ (line 15). Otherwise, we invoke SWEEP procedure to sweep vertices using the sweep rules introduced in Section 5.5. We will introduce the SWEEP procedure in detail later.

In phase 2 (lines 16-21), we first check whether the source vertex u is a strong side-vertex. If so, we skip phase 2 since a strong side-vertex is not contained in any vertex cut with size smaller than k. Otherwise, we perform pair-wise local connectivity testings for all vertices in $N(u)$. Here, we apply the *group sweep rule 3* and skip testing those pairs of vertices that are in the same side-group.

Procedure SWEEP**.** The procedure SWEEP is shown in Algorithm 23. To sweep a vertex v, we set $pru(v)$ to be true. This operation may result in neighbor sweep and group sweep of other vertices as follows.

- (Neighbor Sweep) In lines 1-5, we consider the neighbor sweep. For all the neighbors w of v that have not been swept, we first increase $deposit(w)$ by 1 based on Definiton 25. Then we consider two cases. The first case is that v is a strong side-vertex. According to *neighbor sweep rule 1* in

Algorithm 23 SWEEP($v, pru, deposit, g\text{-}deposit, \mathcal{CS}$)

```
1: pru(v) ← true;
2: for each w ∈ N(v) s.t. pru(w) = false do
3:    deposit(w)++;
4:    if v is a strong side-vertex or deposit(w) ≥ k then
5:       SWEEP(w, pru, deposit, g-deposit, CS);
6: if v is contained in a CC_i and CC_i has not been processed then
7:    g-deposit(CC_i)++;
8:    if v is a strong side-vertex or g-deposit(CC_i) ≥ k then
9:       mark CC_i as processed;
10:      for each w ∈ CC_i s.t. pru(w) = false do
11:         SWEEP(w, pru, deposit, g-deposit, CS)
```

Section 5.5.1, w can be swept since w is a neighbor of v. The second case is $deposit(w) > k$. According to *neighbor sweep rule 2* in Section 5.5.1, w can be swept. In both cases, we invoke SWEEP to sweep w recursively (lines 4-5).

- (Group Sweep) In lines 6-11, we consider the group sweep if v is contained in a side-group $\mathcal{CC}_i$. We first increase $g\text{-}deposit(\mathcal{CC}_i)$ by 1 based on Definiton 27. Then we consider two cases. The first case is that v is a strong side-vertex. According to *group sweep rule 1* in Section 5.5.2, we can sweep all vertices in $\mathcal{CC}_i$. The second case is that $g\text{-}deposit \geq k$. According to *group sweep rule 2* in Section 5.5.2, we can sweep all vertices in $\mathcal{CC}_i$. In both cases we recursively invoke SWEEP to sweep each unswept vertex in $\mathcal{CC}_i$ (lines 8-11).

5.6 Performance Studies

In this section, we experimentally evaluate the performance of our proposed algorithms. We use VCCE to denote the basic implementation of the framework KVCC-ENUM; it invokes GLOBAL-CUT (Algorithm 21) to compute a vertex cut.

We use VCCE* to denote our final optimized algorithm with both neighbor sweep and group sweep strategies. In contrast to VCCE, VCCE* invokes GLOBAL-CUT* which is given in Algorithm 22.

All algorithms are implemented in C++ using gcc complier at -O3 optimization level. All the experiments are conducted under a Linux operating system running on a machine with an Intel Xeon 3.4GHz CPU, 32GB 1866MHz DDR3-RAM. The time cost of algorithms is measured as the amount of wall-clock time elapsed during program execution.

Datasets	vertices	edges	$\overline{d}$
ca-CondMat	23,133	93,497	8.08
ca-AstroPh	18,772	198,110	21.11
Stanford	281,903	2,312,497	16.41
cnr	325,557	3,216,152	19.76
DBLP	986,324	6,707,236	13.60
Web-BerkStan	685,230	7,600,595	22.18
as-Skitter	1,696,415	11,095,298	13.08
Cit	3,774,768	16,518,948	8.75
LiveJournal	4,847,571	68,993,773	28.47
Webbase	118,142,155	1,019,903,190	17.27

Table 5.1: Network statistics

Datasets. We evaluate algorithms on 10 publicly available real-world networks, collaboration network of Arxiv Condensed Matter (ca-CondMat), collaboration network of Arxiv Astro Physics (ca-AstroPh), web graph of Stanford.edu (Stanford), web graph of Italian CNR domain (cnr), DBLP collaboration network (DBLP), web graph of Berkeley and Stanford (Web-BerkStan), Internet topology graph (as-Skitter), citation network among US Patents (Cit), LiveJournal online social network (LiveJournal), and web graph form WebBase crawler (Webbase). The detailed statistics are shown in Table 5.1. The networks are displayed in non-decreasing order regarding the number of edges. All networks and corre-

sponding detailed description can be found in SNAP[1] and Webgraph[2].

Parameter Setting. Given a graph G, let $k_{max}(G)$ be the maximum k such that a k-VCC exists. For the input parameter k, we choose 20%, 40%, 60%, and 80% of the $k_{max}(G)$ of each tested graph G, with $k = 40\% \cdot k_{max}(G)$ as default.

5.6.1 Performance Studies on Real-World Graphs

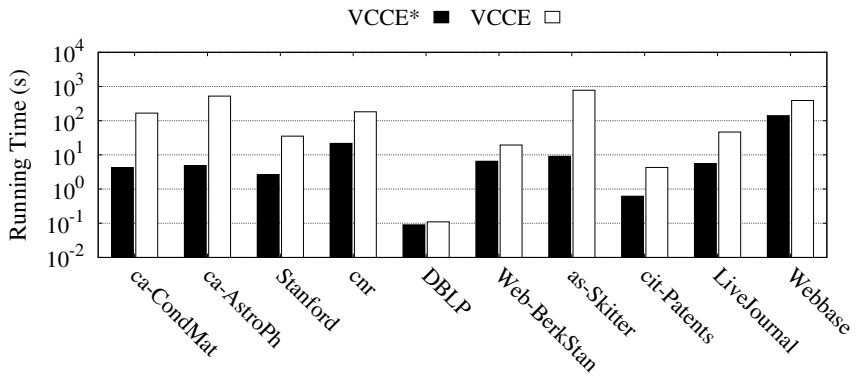


Figure 5.5: Performance on different datasets

We report the running time of VCCE* on all datasets with VCCE as a comparison in Fig. 5.5. The efficiency of VCCE* depends on the graph structure and the input parameter k. We can find that VCCE* is significantly faster than VCCE on most of datasets. It is the widest that the gap between the bars of two algorithms on ca-AstroPh; VCCE* costs about 4 seconds, while VCCE costs over 500 seconds. Note that the algorithmic speedup on DBLP is not obvious. In this case, a large number of vertices are removed due to the k-core constraint and the number of k-VCCs in DBLP is small under the default parameter setting. Even though about 45% of LOC-CUT invocations are avoided in VCCE* due to our pruning techniques, the total number of LOC-CUT invocations is quite small

[1] `http://snap.stanford.edu/index.html`
[2] `http://webgraph.di.unimi.it/`

(about 100) in VCCE. When the parameter k drops to 20% of k_{max} on DBLP, VCCE costs about 80 seconds, while VCCE* costs only about 4 seconds. The detailed relationship between the algorithmic efficiency and the parameter k will be analyzed later. The most time consuming of VCCE* appears on Webbase, which is the largest dataset in the experiments. VCCE* costs about 140 seconds to compute all k-VCCs in Webbase, while VCCE costs about 390 seconds.

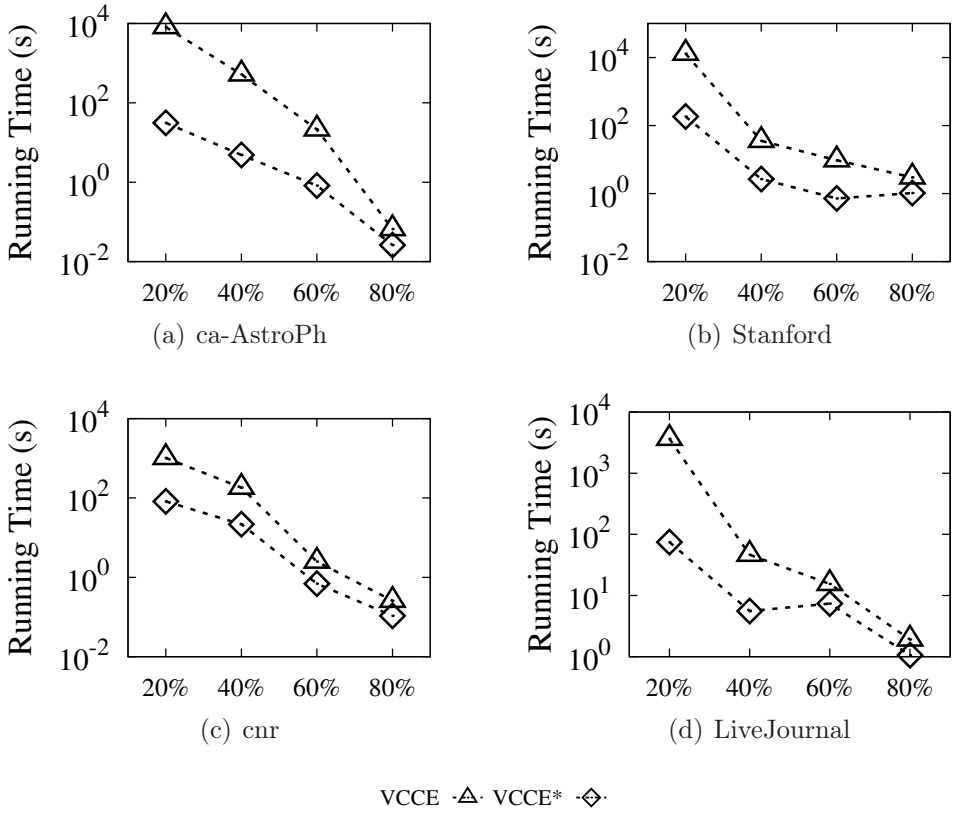


Figure 5.6: Against basic algorithm (vary k)

Vary k. We report the runing time of our proposed algorithms on different datasets when varying k in Fig. 5.6. Due to the space limitation, we only show the results on ca-AstroPh, Stanford, cnr and LiveJournal, while the results on other datasets have similar trends.

The running time of VCCE presents a downward trend on all datasets when raising k. For example, in Fig. 5.6 (c), VCCE costs about 17 minutes to compute all k-VCCs in cnr when $k = 20\% \cdot k_{max}$; the running time drops to under 1 second given $k = 80\% \cdot k_{max}$. The decrease of running time of VCCE is mainly due to following two reasons. First, given a large value of parameter k, a large number of vertices are removed due to the k-core constraint (line 2 of Algorithm 20). Second, it is more likely to find a qualified vertex cut (line 15 of GLOBAL-CUT) given a high value of parameter k; that means the graph is more likely to be partitioned into small subgraphs.

We can find in Fig. 5.6 that VCCE* is faster than VCCE on all parameter settings and the gap between the running time of two algorithms is wide when k is small. This phenomenon demonstrates that our optimized techqniues is more effective when VCCE requires a great amount of computational cost. For example, in Fig. 5.6 (a), VCCE costs over 2 hours when k is 20% of k_{max}. In contrast, VCCE* only need about 30 seconds, which is over 200x faster than VCCE on this parameter setting. The running time of VCCE* and VCCE drops to about 0.3 seconds and 0.7 seconds respectively when $k = 80\% \cdot k_{max}$.

Note that there exists a slight increase for the running time of VCCE* from 60% to 80% in Fig. 5.6 (d). This is because both of the numbers of strong side-vertices and vertices in side-groups decrease when raising k. Even though the total number of max-flow computations in VCCE decreases, the ratio of prunned computations increases and this leads to a compromise result. Similar phenomenon also appears from 60% to 80% in Fig. 5.6 (b).

5.6.2 Evaluating Optimization Techniques

To further investigate the effectiveness of our sweep rules, we also track each processed vertex during the performance of VCCE* and record the number of vertices

Input Parameter	*Non-Pru*	*NS_1*	*NS_2*	*GS*
ca-AstroPh				
$20\% \cdot k_{max}$	3.5%	20.4%	30.2%	45.9%
$40\% \cdot k_{max}$	3.8%	40.0%	23.8%	32.3%
$60\% \cdot k_{max}$	13.8%	42.10%	38.40%	5.70%
$80\% \cdot k_{max}$	100%	0%	0%	0%
avg	4.1%	27.3%	28.7%	39.9%
Stanford				
$20\% \cdot k_{max}$	2.9%	32.3%	14.0%	50.9%
$40\% \cdot k_{max}$	10.1%	12.0%	18.7%	59.2%
$60\% \cdot k_{max}$	7.1%	3.6%	25.6%	63.6%
$80\% \cdot k_{max}$	12.0%	0%	14.7%	73.3%
avg	3.3%	30.7%	14.4%	51.6%
cnr				
$20\% \cdot k_{max}$	3.0%	12.4%	16.0%	68.6%
$40\% \cdot k_{max}$	30.7%	8.9%	47.3%	13.2%
$60\% \cdot k_{max}$	15.8%	27.1%	16.3%	40.7%
$80\% \cdot k_{max}$	61.1%	38.9%	0%	0%
avg	7.0%	12.1%	20.4%	60.5%
LiveJournal				
$20\% \cdot k_{max}$	1.5%	3.6%	39.2%	55.7%
$40\% \cdot k_{max}$	8.5%	6.9%	61.1%	23.4%
$60\% \cdot k_{max}$	19.3%	2.6%	69.7%	8.4%
$80\% \cdot k_{max}$	84.4%	15.6%	0%	0%
avg	3.4%	4.0%	43.2%	49.4%

Table 5.2: Evaluating pruning rules

pruned by each strategy. Specifically, when performing sweep procedure, we separately mark the vertices pruned by *neighbor sweep rule 1* (strong-side vertex), *neighbor sweep rule 2* (neighbor deposit) and group sweep. Here, we divide neighbor sweep into two detailed sub-rules since the both of them perform well and the effectiveness of these two strategies is not very consistent in different datasets. For each vertex v in line 11 of GLOBAL-CUT*, we increase the count for corresponding strategy if v is pruned (line 12). We also record the number of

vertices which are non-pruned and really tested (line 13). For each dataset, we still record these staistics on different k from 20% to 80% of k_{max}. Additionally, we sum the processed vertices on all parameter settings and calculate the average raio of each prunning rule for each dataset. All results are summarized in Table 5.2. *NS_1* and *NS_2* represent *neighbor sweep rule 1* and *neighbor sweep rule 2* respectively. *GS* is group sweep and *Non-Pru* means the proportion of non-pruned vertices. If more than one rules can be adopted to prune a vertex, we mark this vertex by the rule following the priority, NS_1, NS_2 and GS.

The result shows our pruning strategies are effective, especially when k is small. The averge ratio of non-pruned vertices is less than 10% on all datasets, and when $k = 20\% \cdot k_{max}$, the ratio of non-pruned vertices is under 5% on all tested datasets and even drops to about 1.5% on LiveJournal. The raio of non-pruned vertices roughly presnets an upward trend on most of datasets. For example, on ca-AstroPh, that ratio reaches 3.8% and 13.8% on $40\% \cdot k_{max}$ and $60\% \cdot k_{max}$ respectively. We can see that there exists no pruned vertex when $k = 80\% \cdot k_{max}$ on ca-AstroPh, because the number of max-flow computations is already very small and the room for improvement is not enough. On ca-AstroPh, VCCE* computes max-flow only 1 time on $80\% \cdot k_{max}$.

The effectiveness of differnet rules depends on the detailed graph strucutre. We can find that the average ratio of group sweep rules is the largest on all datasets. It accounts for about 39.9% of total on ca-AstroPh and up to 60.5% on cnr. The *neighbor sweep rule 1* performs well on Stanford; its average ratio is about 30.7%. However, this rule only prunes about 4% of total on LiveJournal. By contrast, the average ratio of *neighbor sweep rule 2* is about 14.4% on Stanford, but up to 43.2% on LiveJournal.

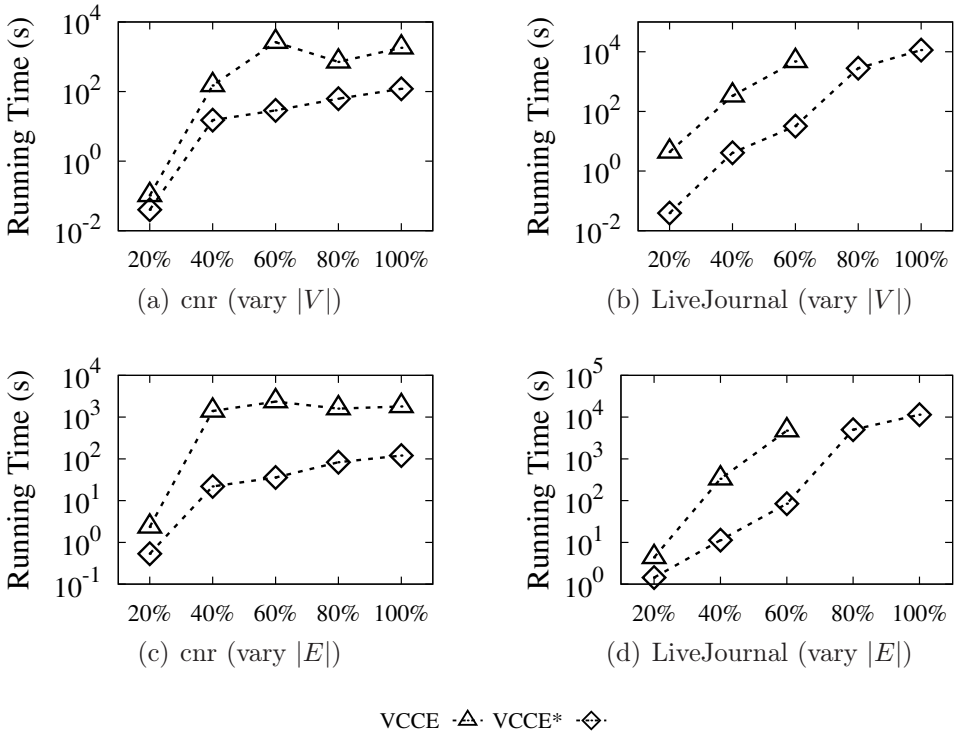


Figure 5.7: Scalability testing

5.6.3 Scalability Testing

In this section, we test the scalability of our proposed algorithms. We choose two real graph datasets cnr and LiveJournal as representatives. For each dataset, we vary the graph size and graph density by randomly sampling vertices and edges respectively from 20% to 100%. When sampling vertices, we get the induced subgraph of the sampled vertices, and when sampling edges, we get the incident vertices of the edges as the vertex set. About the parameter setting, we fix k to 10% of k_{max} of the original graph for all sampling ratio on each dataset. The experimental results are shown in Fig. 5.7.

Fig. 5.7 (a) and (b) report the processing time of our proposed algorithms when varying $|V|$ in cnr and LiveJournal respectively. The curves in Fig. 5.7

(c) and (d) report the processing time of our algorithms on cnr and LiveJournal respectively when varying $|E|$. Note that in Fig. 5.7 (b) and (d), the running time of VCCE is not given on 60% and 80%, since the corresponding procedure cannot finish in 24 hours.

The efficiency of VCCE heavily relies on the detail graph structures, and the lines of VCCE are not stable. For example, in Fig. 5.7 (a), the running time of VCCE is about 3 seconds when sampling 20% nodes, and it reaches about 40 minutes on 60%. Then running time drops slightly to about 27 minutes on 80%, and increase back to about 30 minutes on 100%. By contrast, the running time of VCCE* presents a steadily upward trend on all datasets when the sampling ratio increases. For example, VCCE* costs less than 0.1 second on 20% in Fig. 5.7 (a); its running time reaches about 15 seconds, 28 seconds and 62 seconds on 40%, 60% and 80% respectively. VCCE* costs about 2 minutes on 100%, which is 15x faster than VCCE. The result shows that our pruning strategies are effective and our optimized algorithm is more efficient and scalable than the basic algorithm.

5.7 Chapter Summary

Computing all k-vertex connected components is a foundational problem and has been studied recently. The state-of-the-art solution does not provide any theoretical guarantee for the running time, and requires high computational cost on computing the vertex cut. In this chapter, we study the problem of k-VCC enumeration and prove that the algorithm terminates in polynomial time. We propose several optimization strategies to significantly improve the efficiency of the algorithm. We conduct extensive experiments using ten real datasets to demonstrate the efficiency of our approach.

Chapter 6

EPILOGUE

In this thesis, we study the community detection problem in large graphs, which has a large number of applications. We propose an efficient index-based algorithm for structural graph clustering. The index size is bounded by $O(m)$, where m is the number of edges in the graph. We also propose an algorithm to maintain the index in dynamic graphs. In addition to graph clustering, we also study the cohesive subgraph detection. For the degree-constrained cohesive subgraph metric (k-core), we propose an I/O efficient semi-external algorithm for core decomposition in web-scale graphs. An I/O efficient core maintenance algorithm is also proposed. For the connectivity-constrained cohesive subgraph metric (k-VCC), we follow a partition-based framework and propose an optimized algorithm to compute all k-VCCs in a given graph. We conduct extensive performance studies on several large real-world datasets to show the efficiency and effectiveness of our approaches.

Based on the research works in this thesis, several future research directions are opened. For instance, an I/O efficient or distributed algorithm for the SCAN model can be designed for the case that the graph cannot be entirely loaded in the main memory of single machine. In addition, an index-based solution for

computing all k-VCCs can be studied, since the k-VCCs should be computed from scratch for each new integer k based on the method proposed in this thesis.

BIBLIOGRAPHY

[1] A. Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9), 1988.

[2] M. Altaf-Ul-Amine, K. Nishikata, T. Korna, T. Miyasato, Y. Shinbo, M. Arifuzzaman, C. Wada, M. Maeda, T. Oshima, H. Mori, and S. Kanaya. Prediction of protein functions based on k-cores of protein-protein interaction networks and amino acid sequences. *Genome Informatics*, 14, 2003.

[3] J. I. Alvarez-Hamelin, L. Dall'Asta, A. Barrat, and A. Vespignani. k-core decomposition: a tool for the visualization of large scale networks. *CoRR*, abs/cs/0504107, 2005.

[4] J. I. Alvarez-Hamelin, L. Dall'Asta, A. Barrat, and A. Vespignani. Large scale networks fingerprinting and visualization using the k-core decomposition. In *NIPS*, 2005.

[5] J. I. 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*, volume 1, 2006.

[6] R. Andersen and K. Chellapilla. Finding dense subgraphs with size bounds. In *Algorithms and Models for the Web-Graph*. 2009.

[7] G. Bader and C. Hogue. An automated method for finding molecular complexes in large protein interaction networks. *BMC Bioinformatics*, 4(1), 2003.

[8] B. Balasundaram, S. Butenko, and I. V. Hicks. Clique relaxations in social network analysis: The maximum k-plex problem. *Operations Research*, 59(1), 2011.

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

[10] D. Berlowitz, S. Cohen, and B. Kimelfeld. Efficient enumeration of maximal k-plexes. In *SIGMOD*, 2015.

[11] F. Bonchi, F. Gullo, A. Kaltenbrunner, and Y. Volkovich. Core decomposition of uncertain graphs. In *KDD*, 2014.

[12] D. Bortner and J. Han. Progressive clustering of networks using structure-connected order of traversal. In *ICDE*, 2010.

[13] D. Bortner and J. Han. Progressive clustering of networks using structure-connected order of traversal. In *ICDE*, pages 653–656, 2010.

[14] L. Caccetta and W. Smyth. Graphs of maximum diameter. *Discrete mathematics*, 102(2):121–141, 1992.

[15] L. Chang, W. Li, X. Lin, L. Qin, and W. Zhang. pscan: Fast and exact structural graph clustering. In *ICDE*, pages 253–264, 2016.

[16] L. Chang, J. X. Yu, and L. Qin. Fast maximal cliques enumeration in sparse graphs. *Algorithmica*, 66, 2013.

[17] L. Chang, J. X. Yu, L. Qin, X. Lin, C. Liu, and W. Liang. Efficiently computing k-edge connected components via graph decomposition. In *SIGMOD*, 2013.

[18] M. Charikar. Greedy approximation algorithms for finding dense components in a graph. In *Proc. of the Third International Workshop on Approximation Algorithms for Combinatorial Optimization*, 2000.

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

[20] J. Cheng, Y. Ke, A. W.-C. Fu, J. X. Yu, and L. Zhu. Finding maximal cliques in massive networks by h*-graph. In *SIGMOD*, pages 447–458, 2010.

[21] J. Cheriyan, M. Kao, and R. Thurimella. Scan-first search and sparse certificates: An improved parallel algorithms for k-vertex connectivity. *SIAM J. Comput.*, 22(1):157–174, 1993.

[22] N. Chiba and T. Nishizeki. Arboricity and subgraph listing algorithms. *SICOMP*, 14(1):210–223, 1985.

[23] J. Cohen. Trusses: Cohesive subgraphs for social network analysis. *National Security Agency Technical Report*, page 16, 2008.

[24] W. Cui, Y. Xiao, H. Wang, and W. Wang. Local search of communities in large graphs. In *SIGMOD*, 2014.

[25] C. H. Ding, X. He, H. Zha, M. Gu, and H. D. Simon. A min-max cut algorithm for graph partitioning and data clustering. In *ICDM*, pages 107–114, 2001.

[26] S. N. Dorogovtsev, A. V. Goltsev, and J. F. F. Mendes. K-core organization of complex networks. *Physical review letters*, 96(4), 2006.

[27] J. Edachery, A. Sen, and F. Brandenburg. Graph clustering using distance-k cliques. In *GD*, 1999.

[28] M. Eidsaa and E. Almaas. S-core network decomposition: A generalization of k-core analysis to weighted networks. *Physical Review E*, 88, 2013.

[29] A. H. Esfahanian and S. Louis Hakimi. On computing the connectivities of graphs and digraphs. *Networks*, 14(2):355–366, 1984.

[30] S. Even. An algorithm for determining whether the connectivity of a graph is at least k. *SIAM J. Comput.*, 4, 1975.

[31] S. Even and R. E. Tarjan. Network flow and testing graph connectivity. *SIAM J. Comput.*, 4, 1975.

[32] M. Farach-Colton and M.-T. Tsai. Computing the degeneracy of large graphs. In *Latin American Symposium on Theoretical Informatics*, 2014.

[33] Z. Galil. Finding the vertex connectivity of graphs. *SIAM J. Comput.*, 9, 1980.

[34] C. Giatsidis, F. D. Malliaros, D. M. Thilikos, and M. Vazirgiannis. Corecluster: A degeneracy based graph clustering framework. In *AAAI*, 2014.

[35] C. Giatsidis, D. M. Thilikos, and M. Vazirgiannis. D-cores: Measuring collaboration of directed graphs based on degeneracy. In *Proc. of ICDM'11*, 2011.

[36] C. Giatsidis, D. M. Thilikos, and M. Vazirgiannis. Evaluating cooperation in communities with the k-core structure. In *ASONAM'11*, 2011.

[37] R. Guimera and L. A. N. Amaral. Functional cartography of complex metabolic networks. *Nature*, 433(7028):895, 2005.

[38] J. Healy, J. Janssen, E. Milios, and W. Aiello. Characterization of graphs using degree cores. In *Algorithms and Models for the Web-Graph*. 2008.

[39] J. Huang, H. Sun, J. Han, H. Deng, Y. Sun, and Y. Liu. Shrink: a structural clustering algorithm for detecting hierarchical communities in networks. In *CIKM*, pages 219–228, 2010.

[40] J. Huang, H. Sun, Q. Song, H. Deng, and J. Han. Revealing density-based clustering structure from the core-connected tree of a network. *TKDE*, 25(8), 2013.

[41] S. Janson and M. J. Luczak. A simple solution to the k-core problem. *Random Struct. Algorithms*, 30(1-2), 2007.

[42] P. Jiang and M. Singh. Spici: a fast clustering algorithm for large biological networks. *Bioinformatics*, 26(8):1105–1111, 2010.

[43] U. Kang and C. Faloutsos. Beyond'caveman communities': Hubs and spokes for graph compression and mining. In *ICDM*, pages 300–309, 2011.

[44] W. Khaouid, M. Barsky, V. Srinivasan, and A. Thomo. K-core decomposition of large networks on a single pc. *PVLDB*, 2015.

[45] V. E. Lee, N. Ruan, R. Jin, and C. Aggarwal. A survey of algorithms for dense subgraph discovery. In *Managing and Mining Graph Data*, pages 303–336. 2010.

[46] R. Li, L. Qin, J. X. Yu, and R. Mao. Influential community search in large networks. *PVLDB*, 2015.

[47] R. Li, J. X. Yu, and R. Mao. Efficient core maintenance in large dynamic graphs. *TKDE*, 26(10), 2014.

[48] Y. Li, Y. Zhao, G. Wang, F. Zhu, Y. Wu, and S. Shi. Effective k-vertex connected component detection in large-scale networks. In *DASFAA*, pages 404–421, 2017.

[49] D. R. Lick and A. T. White. k-degenerate graphs. *CJM*, 22, 1970.

[50] S. Lim, S. Ryu, S. Kwon, K. Jung, and J.-G. Lee. Linkscan*: Overlapping community detection using the link-space transformation. In *ICDE*, pages 292–303, 2014.

[51] Y. Liu, J. Lu, H. Yang, X. Xiao, and Z. Wei. Towards maximum independent sets on massive graphs. *PVLDB*, 8(13), 2015.

[52] R. D. Luce. Connectivity and generalized cliques in sociometric group structure. *Psychometrika*, 15, 1950.

[53] T. Luczak. Size and connectivity of the k-core of a random graph. *Discrete Math.*, 91(1), 1991.

[54] S. T. Mai, M. S. Dieu, I. Assent, J. Jacobsen, J. Kristensen, and M. Birk. Scalable and interactive graph clustering algorithm on multicore cpus. In *ICDE*, pages 349–360, 2017.

[55] D. W. Matula. k-blocks and ultrablocks in graphs. *JCTB*, 24(1), 1978.

[56] D. W. Matula and L. L. Beck. Smallest-last ordering and clustering and graph coloring algorithms. *J. ACM*, 30(3), 1983.

[57] K. Menger. Zur allgemeinen kurventheorie. *Fundamenta Mathematicae*, 10(1):96–115, 1927.

[58] R. J. Mokken. Cliques, clubs and clans. *Quality and Quantity*, 13, 1979.

[59] M. Molloy. Cores in random hypergraphs and boolean formulas. *Random Struct. Algorithms*, 27(1), 2005.

[60] A. Montresor, F. De Pellegrini, and D. Miorandi. Distributed k-core decomposition. *TPDS*, 24(2), 2013.

[61] J. Moody and D. R. White. Structural cohesion and embeddedness: A hierarchical conception of social groups. *American Sociological Review*, 68, 2000.

[62] M. E. Newman. Modularity and community structure in networks. *PNAS*, 103(23), 2006.

[63] M. E. Newman and M. Girvan. Finding and evaluating community structure in networks. *Physical review E*, 69(2):026113, 2004.

[64] M. P. O'Brien and B. D. Sullivan. Locally estimating core numbers. In *ICDM*, 2014.

[65] B. Pittel, J. Spencer, and N. Wormald. Sudden emergence of a giant k-core in a random graph. *J. Comb. Theory Ser. B*, 67(1), 1996.

[66] L. Qin, R. Li, L. Chang, and C. Zhang. Locally densest subgraph discovery. In *KDD*, 2015.

[67] F. Radicchi, C. Castellano, F. Cecconi, V. Loreto, and D. Parisi. Defining and identifying communities in networks. *PNAS*, 101(9), 2004.

[68] A. E. Saríyüce, B. Gedik, G. Jacques-Silva, K.-L. Wu, and U. V. Çatalyürek. Streaming algorithms for k-core decomposition. *PVLDB*, 6(6), 2013.

[69] S. B. Seidman. Network structure and minimum degree. *Social networks*, 5(3), 1983.

[70] S. B. Seidman and B. L. Foster. A graph-theoretic generalization of the clique concept. *Journal of Mathematical Sociology*, 6, 1978.

[71] Y. Shao, L. Chen, and B. Cui. Efficient cohesive subgraphs detection in parallel. In *SIGMOD*, 2014.

[72] J. Shi and J. Malik. Normalized cuts and image segmentation. *TPAMI*, 22(8):888–905, 2000.

[73] H. Shiokawa, Y. Fujiwara, and M. Onizuka. Fast algorithm for modularity-based graph clustering. In *AAAI*, pages 1170–1176, 2013.

[74] H. Shiokawa, Y. Fujiwara, and M. Onizuka. Scan++: efficient algorithm for finding clusters, hubs and outliers on large-scale graphs. *PVLDB*, 8(11):1178–1189, 2015.

[75] R. S. Sinkovits, J. Moody, B. T. Oztan, and D. R. White. Fast determination of structurally cohesive subgroups in large networks. *JoCS*, 17, 2016.

[76] M. Sozio and A. Gionis. The community-search problem and how to plan a successful cocktail party. In *KDD*, 2010.

[77] H. Sun, J. Huang, J. Han, H. Deng, P. Zhao, and B. Feng. gskeletonclu: Density-based network clustering via structure-connected tree division or agglomeration. In *ICDM*, pages 481–490. IEEE, 2010.

[78] J. Torrents and F. Ferraro. Structural cohesion: Visualization and heuristics for fast computation. *JoSS*, 16:1–36, 2015.

[79] A. Verma and S. Butenko. Network clustering via clique relaxations: A community based approach. *Graph Partitioning and Graph Clustering*, 588, 2012.

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

[81] L. Wang, Y. Xiao, B. Shao, and H. Wang. How to partition a billion-node graph. In *ICDE*, 2014.

[82] M. Wang, C. Wang, J. X. Yu, and J. Zhang. Community detection in social networks: an in-depth benchmarking study with a procedure-oriented framework. *PVLDB*, 8(10), 2015.

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

[84] D. Wen, L. Qin, X. Lin, Y. Zhang, and L. Chang. Enumerating k-vertex connected components in large graphs. *arXiv:1703.08668*, 2017.

[85] D. Wen, L. Qin, Y. Zhang, L. Chang, and X. Lin. Efficient structural graph clustering: An index-based approach. *PVLDB*, 11(3), 2017.

[86] D. Wen, L. Qin, Y. Zhang, X. Lin, and J. X. Yu. I/O efficient core graph decomposition at web scale. In *ICDE*, 2016.

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

[88] H. Whitney. Congruent graphs and the connectivity of graphs. *American Journal of Mathematics*, 54(1):150–168, 1932.

[89] J. Xie, S. Kelley, and B. K. Szymanski. Overlapping community detection in networks: The state-of-the-art and comparative study. *ACM Comput. Surv.*, 45(4), 2013.

[90] X. Xu, N. Yuruk, Z. Feng, and T. A. J. Schweiger. Scan: A structural clustering algorithm for networks. In *KDD*, pages 824–833, 2007.

[91] X. Yan, X. J. Zhou, and J. Han. Mining closed relational graphs with connectivity constraints. In *ICDE*, 2005.

[92] Z. Zeng, J. Wang, L. Zhou, and G. Karypis. Coherent closed quasi-clique discovery from large dense graph databases. In *KDD*, 2006.

[93] H. Zhang, H. Zhao, W. Cai, J. Liu, and W. Zhou. Using the k-core decomposition to analyze the static structure of large-scale software systems. *The Journal of Supercomputing*, 53(2), 2010.

[94] Z. Zhang, J. X. Yu, L. Qin, L. Chang, and X. Lin. I/O efficient: computing sccs in massive graphs. In *SIGMOD*, 2013.

[95] Z. Zhang, J. X. Yu, L. Qin, and Z. Shang. Divide & conquer: I/O efficient depth-first search. In *SIGMOD*, 2015.

[96] F. Zhao and A. K. Tung. Large scale cohesive subgraphs discovery for social network visual analysis. In *PVLDB*, volume 6, 2012.

[97] W. Zhao, V. Martha, and X. Xu. Pscan: a parallel structural clustering algorithm for big networks in mapreduce. In *AINA*, pages 862–869, 2013.

[98] 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 *Proc. of EDBT'12*, 2012.

[99] Y. Zhou, H. Cheng, and J. X. Yu. Graph clustering based on structural/attribute similarities. *PVLDB*, 2(1):718–729, 2009.