Fast Algorithms for Large-Scale Network Analytics

Dissertation

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Ahmet Erdem Sarıyüce, B.S.

Graduate Program in Computer Science and Engineering

The Ohio State University

2015

Dissertation Committee:

Ümit V. Çatalyürek, Advisor

Arnab Nandi

Srinivasan Parthasarathy

© Copyright by

Ahmet Erdem Sarıyüce

2015

Abstract

Today's networks are massive and dynamic; Facebook with a billion of users and a trillion of connections and Twitter with ∼600 millions of users tweeting ∼9,000 times in a second are just a few examples. Making sense of these graphs in static and dynamic scenarios is essential. Most of the existing algorithms assume that the graph is static and it does not change. Today, these assumptions are no more valid. Fast algorithms for streaming and parallel scenarios are necessary to process graphs of massive sizes. Compression techniques are also quite necessary to deal with the size. In our work, we provide compression, streaming, and parallel algorithms for three important graph analytics problems: centrality computation, dense subgraph discovery and community detection. In addition, we introduce new dense subgraph discovery algorithms to better model the cohesion in real-world networks.

Centrality metrics, such as betweenness and closeness, quantify how central a node is in a network. They have been used successfully to carry various analyses such as structural analysis of knowledge networks, power grid contingency analysis, quantifying importance in social networks, analysis of covert networks and decision/action networks, and even for finding the best store locations in cities. However, they are

computationally expensive kernels. We present two different approaches for speeding up the centrality computation. First, we propose the framework, BADIOS, which compresses a network and shatters it into pieces so that the betweenness and closeness centrality computations can be handled independently for each piece. Second, we show how centrality computations can be regularized to reach higher performance on cutting edge hardware. Last, but not least, we provide incremental algorithms to efficiently maintain closeness centrality values of vertices upon edge changes in the graphs.

Finding dense subgraphs is a critical aspect of graph mining. It has been used for finding communities and spam link farms in web graphs, graph visualization, real-time story identification, DNA motif detection in biological networks, finding correlated genes, epilepsy prediction, finding price value motifs in financial data, graph compression, distance query indexing, and increasing the throughput of social networking site servers. Motivated by the dynamic nature of graphs, we introduce incremental algorithms for k-core decomposition, which is proven to be a fast and effective solution for dense subgraph discovery problem. Furthermore, we present new algorithms to find high-quality dense subgraphs and the relations among them in networks. To this end, we introduce nucleus decomposition of a graph, which represents the graph as a forest of nuclei and results in denser subgraphs than the state-of-the-art methods.

Community detection is a fundamental analytic in graph processing that can be applied to several application domains, such as social networks. In this context, communities are often overlapping, as a person can be involved in more than one community. We address the problem of streaming overlapping community detection, where the goal is to incrementally maintain communities in the presence of streaming updates.

To my family

Acknowledgments

I would like to thank to my advisor, Ümit V. Çatalyürek, for his guidance and support during my doctoral study.

I am also thankful to my committee members, Arnab Nandi and Srinivasan Parthasarathy, for spending their time and effort to read and comment on my dissertation.

I was fortunate to work with great people in my Ph.D. Erik Saule has been a great mentor during my first years, and I learned a lot from him. Kamer Kaya has been an awesome friend, collaborator, and co-eater to me. I was very lucky to meet with Buğra Gedik, he was inspiring to me and I owe him a lot. Gabriela Jacques-Silva and Kun-Lung Wu were great mentors at my IBM Research internships and I will not forget their encouragements. Lastly, I thank to Ali Pınar and C. Seshadhri for their help and collaboration during my Sandia internship.

I had great friends in Columbus. I am thankful to my comrade and lab-mate Mehmet Deveci and great friend Ali Adalı.

Ph.D. is a long and tiring process, and I am still not sure if it did worth to spend those years for this purpose. My family has been with me from the beginning of this

journey and I am indebted to God for having them. My mom, Safiye, and my dad, Bilal, gave me endless support and I dedicate this dissertation to them. I also thank my brothers, Emrah, Emirhan, and Abdullah, and my sister-in-law Merve for being with me.

Last, but not least, I thank to my fiancée Büşra for being my other half.

Vita

2010 ..... B.S. Computer Engineering,
Middle East Technical University.

2010 – present ..... Graduate Research Associate,
Computer Science and Engineering,
The Ohio State University.

Publications

Research Publications

Ahmet Erdem Sarıyüce, Erik Saule, and Ümit V. Çatalyürek. Improving Graph Coloring on Distributed Memory Parallel Computers. In *International Conference on High Performance Computing (HiPC)*, Dec 2011.

Ahmet Erdem Sarıyüce, Erik Saule, and Ümit V. Çatalyürek. Scalable Hybrid Implementation of Graph Coloring using MPI and OpenMP. In *Workshop on Parallel Computing and Optimization (PCO), in conjunction with IPDPS,* , May 2012.

Ahmet Erdem Sarıyüce, Kamer Kaya, Erik Saule, and Ümit V. Çatalyürek. Betweenness centrality on gpus and heterogeneous architectures. In *Workshop on General Purpose Processing Using GPUs (GPGPU), in conjunction with ASPLOS*, Mar 2013.

Ahmet Erdem Sarıyüce, Erik Saule, Kamer Kaya, and Ümit V. Çatalyürek. Shattering and compressing networks for betweenness centrality. In *SIAM International Conference on Data Mining, (SDM)*, May 2013.

Ahmet Erdem Sarıyüce, Buğra Gedik, Gabriela Jacques-Silva, Kun-Lung Wu and Ümit V. Çatalyürek. Streaming algorithms for k-core decomposition. In *International Conference on Very Large Data Bases (VLDB)*, Aug 2013.

Ahmet Erdem Sarıyüce, Erik Saule, Kamer Kaya, and Ümit V. Çatalyürek. Streamer: a distributed framework for incremental closeness centrality computation. In *IEEE Cluster Conference*, Sep 2013.

Ahmet Erdem Sarıyüce, Kamer Kaya, Erik Saule, and Ümit V. Çatalyürek. Incremental algorithms for closeness centrality. In *IEEE International Conference on BigData*, Oct 2013.

Ahmet Erdem Sarıyüce, Erik Saule, Kamer Kaya, and Ümit V. Çatalyürek. Hardware/software vectorization for closeness centrality on multi-/many-core architectures. In *Workshop on Multithreaded Architectures and Applications (MTAAP), in conjunction with IPDPS*, May 2014.

Ahmet Erdem Sarıyüce, Erik Saule, Kamer Kaya, and Ümit V. Çatalyürek. Regularizing graph centrality computations. *Journal of Parallel and Distributed Computing*, 76(0):106 – 119, 2015.

Ahmet Erdem Sarıyüce, Erik Saule, Kamer Kaya, and Ümit V. Çatalyürek. Incremental closeness centrality in distributed memory. *Parallel Computing*, 2015.

Ahmet Erdem Sarıyüce, C. Seshadhri, Ali Pınar, and Ümit V. Çatalyürek. Finding the hierarchy of dense subgraphs using nucleus decompositions. In *International World Wide Web Conference (WWW)*, May 2015.

Fields of Study

Major Field: Computer Science and Engineering

Table of Contents

Page

List of Tables

Table **Page**

List of Figures

Figure **Page**

Chapter 1: Introduction

Relationships between entities, such as people and systems, can be captured as graphs where vertices represent entities and edges represent connections among them. In many applications, it is highly beneficial to capture this graph structure and analyze it. For instance, in a social network, finding communities in the graph [62] can facilitate targeted advertising. In the web graphs, finding densely connected regions in the graph [55] may help identify link spam [142]. In telecommunications graphs, with call relationships, locating closely connected groups of people for generating promotions is important [128]. In protein-protein interaction graphs, locating cliques in protein structures can be used for comparative modeling and prediction [144].

Many real-world graphs are highly dynamic. In social networks, users join/leave and connections are created/severed on a regular basis. In the web graph, new links are established and severed as a natural result of content update and creation. In customer call graphs, new edges are added as people extend their list of contacts. Furthermore, many applications require analyzing such graphs over a time window, as newly forming relationships may be more important than the old ones. For instance,

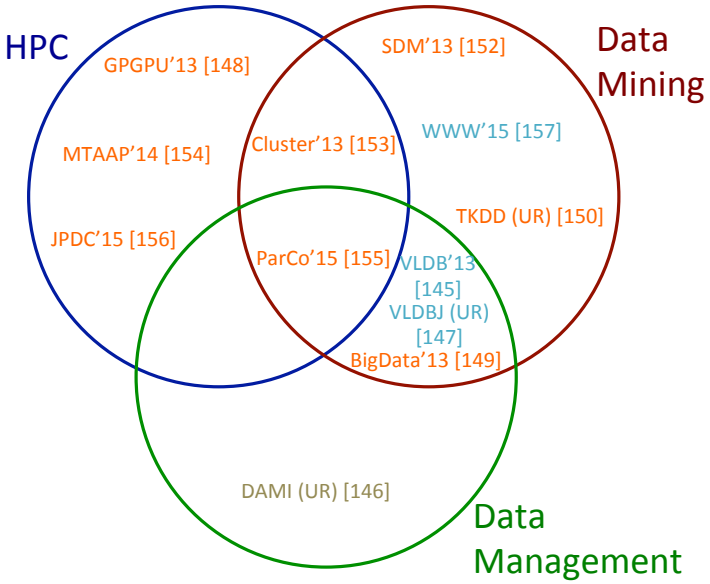


Figure 1.1: Contributions of the dissertation, classified by subdisciplines, and color coded by graph analytics noted below the figure, with the relevant publications. UR: Under Review.

in customer call graphs, the historic calls are not too relevant for churn detection. Looking at a time window naturally brings removals as key operations like insertions. This is because as edges slide out of the time window, they have to be removed from the graph of interest. In summary, dynamic graphs where edges are added and removed continuously are common in practice and represent an important use case.

Main focus of this dissertation is "fast" algorithms. We always aim to reduce the absolute execution time for different algorithms under different settings. In general, we do not reduce asymptotic complexity of an algorithm, instead we focus on effective heuristics to speed up the computation. Furthermore, we make use of cutting edge

hardware for faster computations. For all studies included in this dissertation, we always implement the existing state-of-the-art algorithm in the most efficient manner. Then, we compare our new algorithms with respect to that efficient baselines. Wherever we introduce a new algorithm, we give the absolute time as well as the speedup numbers with respect to the efficient baseline implementation. We believe that the most important thing to evaluate the efficiency of a new algorithm is to have a proper implementation of the baseline. For instance, in Chapter 2, when we compare our fast algorithms for betweenness centrality computation with respect to the literature, we see that many existing implementations are quite slow, and creates an illusion of high speed up numbers. Our implementation of the baseline algorithm for betweenness centrality computation is 40-50 times faster than the fastest algorithm in one of the existing work which claims huge speedups. We also believe that our algorithms are capable to meet the needs of the datasets we have used. For example, in Chapter 5, we show that k-core decomposition of 16M edge graph can be maintained in a very fast rate: 10K edge insertions can be handled in a single **second**, enabling real-time processing at that scale.

In this study, we focus on fast algorithms for various types of graph analytics problems: centrality computation, dense subgraph discovery, and community detection. Figure 1.1 summarizes the focus of this study, classified by the subdisciplines and graph analytics algorithms with all the published and under review work. Subjects of the studies are grouped into three topics; fast and incremental centrality computation,

incremental and high-quality dense subgraph discovery and incremental overlapping community detection. Our contributions span to three important subdisciplines of computer science: high performance computing (HPC), data mining, and data management.

- In the HPC domain, we introduced parallel algorithms for fast centrality computation [148, 153, 154, 155, 156].
- We investigated sliding window streaming algorithms for closeness centrality, k-core decomposition, and overlapping community detection problems, which deal with the management of data. Some of these algorithms are parallel [145, 146, 147, 149, 155].
- Regarding the data mining area, we devised compression algorithms for fast centrality computation, incremental algorithms for closeness centrality and k-core decomposition, and high-quality algorithms for dense subgraph discovery problem [145, 147, 149, 150, 152, 153, 155, 157].

We believe that this dissertation will be beneficial for computer science researchers working on fast algorithms as well as the domain scientists, who are in need of fast algorithms to make sense of the graphs. For the computer science perspective, we believe that our contributions for different graph analytics will long live since significant portion of them are independent of computer architecture, indicating that they can

be used as a baseline for any new algorithm on any given architecture. We also introduce quite efficient parallel algorithms on cutting edge architectures, like GPUs and distributed-memory machines, which have been proven to be inevitable tools for high performance computing. For the application domains, like sociology, bioinformatics, and web science, problems in this dissertation have many applications, explained in detail at the beginning of each chapter, and domain scientists can make use of our algorithms to work on large scale networks in a more efficient manner.

In the following sections, we briefly present the motivation and specific problems we studied in this dissertation. Then, we summarize our contributions for each graph analytic problems and give pointers to associated chapters.

1.1 Fast and Incremental Centrality Computation

Centrality metrics play an important role while detecting the central and influential nodes in various types of networks such as social networks [112], biological networks [99], power networks [92], covert networks [100] and decision/action networks [48]. The *betweenness* and *closeness* metric have always been interesting and have been implemented in several tools which are widely used in practice for analyzing networks and graphs [114]. In short, the betweenness centrality (BC) score of a node is the sum of the fractions of the shortest paths between all node pairs that pass through the node of interest [65] and the closeness centrality (CC) score of a node is the inverse of the sum of shortest distances from the node of interest to all other

nodes. Hence, they are measure of the contribution, load, influence, and effectiveness of a node while disseminating information through a network.

To make the centrality computation faster in sequential settings, we propose the **BADIOS** framework which uses a set of techniques (based on **B**ridges, **A**rticulation, **D**egree-1, and **I**dentical vertices, **O**rdering, and **S**ide vertices) for faster betweenness and closeness centrality computation, in Chapter 2. The framework shatters the network by removing **B**ridges and **A**rticulation points, and reduces its size so that the BC and CC scores of the nodes in different pieces of network can be computed correctly and independently, and hence, in a more efficient manner. **BADIOS** also compresses the graph by removing **D**egree-1 vertices recursively, by eliminating the **I**dentical vertices, which have same neighborhood, and by deleting **S**ide vertices, whose neighborhoods form a clique. Last, but not least, it also preorders the graph (**O**rdering) to improve cache utilization. Details are presented in Chapter 2 and in [150, 152]. In summary, the contribution of this dissertation on this topic are as follows:

- We propose **BADIOS** framework to manipulate graphs by shattering and compressing them for fast centrality computation.
- We present BC and CC algorithms for computing the centrality values on manipulated graphs.

- Proposed algorithms are experimentally evaluated. For one of our social networks, we achieve to reduce the BC computation time from 5 days to 16 hours and CC computation time from 3 days to 6 hours.

Huge computational cost of the centrality algorithms necessitates the leveraging of the cutting edge hardware. In Chapter 3, we show how centrality computations can be regularized to reach higher performance. For betweenness centrality, we deviate from the traditional fine-grain approach by allowing a GPU to execute multiple breadth-first searches (BFSs) at the same time. Furthermore, we exploit hardware and software vectorization to compute closeness centrality values on CPUs, GPUs and Intel Xeon Phi. Chapter 3 introduces our study on this topic, and more information can be found in [148, 154, 156]. Contribution of this study can be summarized as follows:

- We propose simultaneous breadth-first search operations for speeding up the BC and CC computation on cutting-edge hardware.
- For CC, we make use of hardware/software vectorization to be applied on CC computation.
- We extensively evaluated our algorithms and techniques on cutting-edge hardware. In particular, we achieve an improvement of a factor 5.9 on CPU architectures, 70.4 on GPU architectures and 21.0 on Intel Xeon Phi.

Motivated by the dynamic nature of graphs, we investigated streaming algorithms for closeness centrality algorithms in sliding-window scenarios. Aim is to maintain centrality values of vertices when there is an edge insertion or removal in the graph. In Chapter 4, we provide computation filtering techniques for incremental CC computation. Our first contributions in Chapter 4 are incremental algorithms which efficiently update the closeness centralities of vertices upon edge insertions and removals. Compared with the existing algorithms, our algorithms have a low-memory footprint which makes them practical and applicable to very large graphs. On top of the sequential incremental closeness centrality algorithms, we present STREAMER, a framework to efficiently parallelize the incremental CC computation on high-performance clusters. STREAMER employs *DataCutter* [27], our in-house data-flow programming framework for distributed memory systems. The best available algorithm for the offline centrality computation is pleasingly parallel (and scalable if enough memory is available) since it involves n independent executions of the single-source shortest path algorithm [29]. There are several (synchronous and asynchronous) blocks in the online approach and it is not trivial to obtain an efficient parallelization of the incremental algorithm. As our experiments will show, the data-flow programming model and pipelined parallelism are very useful to achieve a significant overlap among these computation/communication blocks and yield a scalable solution for the incremental centrality computation.

Chapter 4, and [149, 153, 155], presents more details on this subject, which can be summarized as follows:

- We introduce the incremental closeness centrality algorithms to maintain centrality values of vertices upon edge changes in the networks.
- We propose the first distributed-memory framework STREAMER for the incremental centrality computation problem which employs a pipelined parallelism to achieve computation-computation and computation-communication overlap.
- We also leverage the shared-memory parallelization and take Non Uniform Memory Architecture (NUMA) effects into account.
- The framework appears to scale linearly: when 63 worker nodes (8 cores/node) are used, for the networks `amazon0601` and `web-Google`, STREAMER obtains 456 and 497 speedups, respectively, compared to a single worker node-single thread execution. Furthermore, using additional techniques provide an improvement of a factor between 2.2 to 9.3 times

1.2 Incremental and High-Quality Dense Subgraph Discovery

Finding dense subgraphs is a critical aspect of graph mining [105]. It has been used for finding communities and spam link farms in web graphs [101, 72, 56], graph visualization [7], real-time story identification [11], DNA motif detection in biological networks [64], finding correlated genes [185], epilepsy prediction [88], finding price

value motifs in financial data [57], graph compression [34], distance query indexing [91], and increasing the throughput of social networking site servers [73]. This is closely related to the classic sociological notion of group cohesion [24, 61]. There are tangential connections to classic community detection, but the objectives are significantly different. Community definitions involve some relation of inner versus outer connections, while dense subgraphs purely focus on internal cohesion.

We study the problem of incrementally maintaining the k-core decomposition of a graph in Chapter 5. A k-core of a graph [161] is a maximal connected subgraph in which every vertex is connected to at least k other vertices. Finding k-cores in a graph is a fundamental operation for many graph algorithms. k-core is commonly used as part of community detection algorithms [70], as well as for finding dense components in graphs [9, 19, 98], as a filtering step for finding large cliques (as a k-clique is also a k-1-core), and for large-scale network visualization [8]. We develop streaming algorithms for k-core decomposition of graphs in sliding-windows scenarios. In particular, we focus on algorithms to update the decomposition as edges are inserted into and removed from the graph (vertex additions and removals are trivial extensions). Details of our contributions on this part of the dissertation is in Chapter 5 and in [145, 147], which can be summarized as follows:

- We develop various algorithms to update the k-core decomposition incrementally. To the best of our knowledge, these are the first such incremental algorithms.

- We identify a small subset of vertices that have to be visited in order to update the density pointer values of vertices in the presence of edge insertions and deletions.

- We present a comparative experimental study that evaluates the performance of our algorithms on real-world and synthetic data sets. Our algorithms provide a significant reduction in run-time compared to non-incremental alternatives, reaching 6 orders of magnitude speedup for a graph of size of around 16 million.

For graph analysis, one rarely looks for just a single (or the optimal, for whatever notion) dense subgraph. We want to find many dense subgraphs and understand the relationships among them. Ideally, we would like to see if they nest within each other, if the dense subgraphs are concentrated in some region, and if they occur at various scales of size and density. Motivated by the following questions:

- How do we attain a global, hierarchical representation of many dense subgraphs in a real-world graph?

- Can we define an efficiently solvable objective that directly provides *many* dense subgraphs? We wish to avoid heuristics, as they can be difficult to predict formally.

In Chapter 6, we present nucleus decomposition [157] for high-quality dense subgraph discovery problem. Our contributions can be summarized as follows:

- Our primary theoretical contribution is the notion of *nuclei* in a graph. Roughly speaking, an (r, s)-nucleus, for fixed (small) positive integers $r < s$, is a maximal

subgraph where every r-clique is part of many s-cliques. (The real definition is more technical and involves some connectivity properties.) Moreover, nuclei that do not contain one another cannot share an r-clique.

- We show that the (r, s)-nuclei (for any $r < s$) form a hierarchical decomposition of a graph. The nuclei are progressively denser as we go towards the leaves in the decomposition. We provide an exact, efficient algorithm that finds all the nuclei for any r, s values and builds the hierarchical decomposition.

- In practice, we observe that $(3, 4)$-nuclei provide the most interesting decomposition. We find the $(3, 4)$-nuclei for a large variety of more than 20 graphs. Our algorithm is feasible in practice, and we are able to process a 39 million edge graph in less than an hour (using commodity hardware).

1.3 Streaming Overlapping Community Detection

Community detection is a fundamental kernel in graph analytics. We can define a community within a graph as a set of vertices that exhibit high *cohesiveness* and low *conductance*. High cohesiveness means that the vertices in the community have relatively high number of edges connecting them, and low conductance means that the vertices in the community have relatively small number of edges going outside of the community.

Communities in social networks have two key characteristics. The first is that communities are *overlapping*, as different communities can have common users. This

is a typical scenario, as a single user can be involved in different communities, such as co-workers, friends, and family. The second is that communities are *dynamic*. They evolve as a result of the continuous interactions between people. These interactions can result in the addition/removal of new/existing relationships in the network. For instance, the follower-followee graph of Twitter [173] is highly active, with millions of updates to the graph structure every day. This number is even higher if we consider the mention graph of Twitter. It is also common to analyze the graph over a recent time window, such as the mention graph of Twitter over the last week. In such scenarios, both insertions and removals are equally frequent.

In Chapter 7, we present SONIC—an algorithm to detect overlapping communities on dynamic graphs in a *streaming* manner. Upon each edge insertion or removal, we *incrementally* maintain the overlapping communities. This way, the communities are updated more efficiently and without the need for periodic re-computations that are typically performed in batch. SONIC maintains multiple community ids for each vertex and updates these ids upon edge insertions and removals. By doing so, it can answer any query for the communities of a given vertex (or a set of vertices) by a simple traversal of the community ids.

More details of our contributions on this part of the dissertation can be found in Chapter 7, and also in [146]. To sum up, major contributions can be listed as follows:

- The SONIC algorithm for incremental overlapping community detection over dynamic graphs with streaming updates.

- A technique to detect significant changes in small community structures to avoid a costly merge, unless a small community change is likely to cause a larger community change.
- Inverted-index and min-hash based techniques to further accelerate the incremental merge used in SONIC.
- An experimental evaluation of SONIC on real-world and synthetic data sets, with respect to quality and running time performance.

Chapter 2: Graph Manipulations for Fast Centrality Computation

Centrality metrics are crucial for detecting the central and influential nodes in various types of networks such as social networks [112], biological networks [99], power networks [92], covert networks [100] and decision/action networks [48]. The *betweenness* and *closeness* are two intriguing metrics and have been implemented in several tools which are widely used in practice for analyzing networks and graphs [114]. The betweenness centrality (BC) score of a node is the sum of the fractions of the shortest paths between node pairs that pass through the node of interest [65], whereas the closeness centrality (CC) score of a node is the inverse of the sum of shortest distances from the node of interest to all other nodes. Hence, contribution, load, influence, or effectiveness of a node, while disseminating information through a network, is determined with betweenness and/or closeness metrics.

Although BC and CC have been proved to be successful for network analysis, computing the centrality scores of all the nodes in a network is expensive. Brandes proposed an algorithm for computing BC with $\mathcal{O}(nm)$ and $\mathcal{O}(nm + n^2 \log n)$ time

complexity and $\mathcal{O}(n+m)$ space complexity for unweighted and weighted networks, respectively, where n is the number of nodes in the network and m is the number of node-node interactions in the network [29]. Brandes' algorithm is currently the best algorithm for BC computations and it is unlikely that general algorithms with better asymptotic complexity can be designed [97]. However, it is not fast enough to handle Facebook's billion or Twitter's 200 million users.

2.1 Introduction

We propose the **BADIOS** framework which uses a set of techniques (based on **B**ridges, **A**rticulation, **D**egree-1, and **I**dentical vertices, **O**rdering, and **S**ide vertices) for faster betweenness and closeness centrality computation. The framework shatters the network and reduces its size so that the BC and CC scores of the nodes in two different pieces of network can be computed correctly and independently, and hence, in a more efficient manner. It also preorders the graph to improve cache utilization.

For the sake of simplicity, we consider only standard, shortest-path vertex-betweenness and vertex-closeness centrality on undirected unweighted graphs. However, our techniques can be used for other path-based centrality metrics, or other BC variants, e.g., *edge* and *group betweenness* [30]. **BADIOS** also applies to weighted and/or directed networks. And all the techniques are compatible with previously proposed approximation and parallelization of the BC and CC computation.

We apply **BADIOS** on a popular set of graphs with sizes ranging from 6K edges to 4.6M edges. For BC, we show an average speedup 2.8 on small graphs and 3.8 on

large ones. In particular, for the largest graph we use, with 2.3M vertices and 4.6M edges, the computation time is reduced from more than 5 days to less than 16 hours. For CC, the average speedup is 2.4 and 3.6 on small and large networks.

The rest of the chapter is organized as follows: In Section 2.2, an algorithmic background for BC and CC computation are given. The shattering and compression techniques are explained in Sections 2.5 and 2.4. Section 2.6 gives experimental results on various kinds of networks. We give the related work in Section 2.7 and summarize the chapter with Section 2.8.

2.2 Notation and Background

Let $G = (V, E)$ be a network modeled as an undirected graph with $n = |V|$ vertices and $m = |E|$ edges where each node is represented by a vertex in V, and a node-node interaction is represented by an edge in E. Let $\Gamma(v)$ be the set of vertices which are interacting with v. A graph $G' = (V', E')$ is a *subgraph* of G if $V' \subseteq V$ and $E' \subseteq E$.

A *path* is a sequence of vertices such that there exists an edge between consecutive vertices. A path between two vertices s and t is denoted by $s \rightsquigarrow t$. Two vertices $u, v \in V$ are *connected* if there is a path from u to v. If this is the case $\texttt{dst}_G(u, v) = \texttt{dst}_G(v, u)$ shows the length of the shortest $u \rightsquigarrow v$ path in G. Otherwise, $\texttt{dst}_G(u, v) = \texttt{dst}_G(v, u) = \infty$. If all vertex pairs are connected we say that G is *connected*. If G is not connected, then it is *disconnected* and each maximal connected subgraph of G is a *connected component*, or a component, of G.

Given a graph $G = (V, E)$, an edge $e \in E$ is a *bridge* if $G - e$ has more number of connected components than G, where $G - e$ is obtained by removing e from E. Similarly, a vertex $v \in V$ is called an *articulation vertex* if $G - v$ has more connected components than G, where $G - v$ is obtained by removing v and its adjacent edges from V and E, respectively. The graph G is *biconnected* if it is connected and it does not contain an articulation vertex. A maximal biconnected subgraph of G is a *biconnected component*: if G is biconnected it has only one biconnected component, which is G itself.

$G = (V, E)$ is a *clique* if and only if $\forall u, v \in V$, $\{u, v\} \in E$. The subgraph *induced by* a subset of vertices $V' \subseteq V$ is $G' = (V', E' = \{V' \times V'\} \cap E)$. A vertex $v \in V$ is a *side vertex* of G if and only if the subgraph of G induced by $\Gamma(v)$ is a clique. Two vertices u and v are *identical* if and only if **either** $\Gamma(u) = \Gamma(v)$ (type-I) **or** $\{u\} \cup \Gamma(u) = \{v\} \cup \Gamma(v)$ (type-II). A vertex v is a *degree-1* vertex if and only if $|\Gamma(v)| = 1$.

2.2.1 Closeness Centrality

Given a graph G, the closeness centrality of u can be defined as

$$\texttt{far}[u] = \sum_{\substack{v \in V \\ \texttt{dst}_G(u,v) \neq \infty}} \texttt{dst}_G(u, v)$$

$$\texttt{cc}[u] = \frac{1}{\texttt{far}[u]}$$

If u cannot reach any vertex in the graph $\texttt{cc}[u] = 0$.

For a sparse unweighted graph $G = (V, E)$ the complexity of CC computation is $\mathcal{O}(n(m+n))$ [29]. The pseudo-code is given in Algorithm 1. For each vertex $s \in V$, the algorithm initiates a breadth-first search (BFS) from s, computes the distances to the other vertices, and accumulates to $\texttt{cc}[s]$. Since a BFS takes $\mathcal{O}(m+n)$ time, and n BFSs are required in total, the complexity follows.

Algorithm 1: CC: Centrality computation kernel

```
   Data: G = (V, E)
   Output: cc[.]
 1 for each s ∈ V do
 2     Q ← empty queue
 3     Q.push(s)
 4     dst[s] ← 0
 5     far ← 0
 6     cc[s] ← 0
 7     dst[v] ← ∞, ∀v ∈ V \ {s}
 8     while Q is not empty do
 9         v ← Q.pop()
10         for all w ∈ Γ_G(v) do
11             if dst[w] = ∞ then
12                 Q.push(w)
13                 dst[w] ← dst[v] + 1
14                 far ← far + dst[w]
15     cc[s] ← 1/far
16 return cc[.]
```

2.2.2 Betweenness Centrality:

Given a connected graph G, let σ_{st} be the number of shortest paths from a source $s \in V$ to a target $t \in V$. Let $\sigma_{st}(v)$ be the number of such $s \rightsquigarrow t$ paths passing

through a vertex $v \in V$, $v \neq s, t$. Let the *pair dependency* of v to s, t pair be the fraction $\delta_{st}(v) = \frac{\sigma_{st}(v)}{\sigma_{st}}$. The betweenness centrality of v is defined by

$$\mathtt{bc}[v] = \sum_{s \neq v \neq t \in V} \delta_{st}(v). \tag{2.1}$$

Since there are $\mathcal{O}(n^2)$ pairs in V, one needs $\mathcal{O}(n^3)$ operations to compute $\mathtt{bc}[v]$ for all $v \in V$ by using (2.1). Brandes reduced this complexity and proposed an $\mathcal{O}(mn)$ algorithm for unweighted networks [29]. The algorithm is based on the accumulation of pair dependencies over target vertices. After accumulation, the dependency of v to $s \in V$ is

$$\delta_s(v) = \sum_{t \in V} \delta_{st}(v). \tag{2.2}$$

Let $\mathtt{P}_s(u)$ be the set of u's predecessors on the shortest paths from s to all vertices in V. That is,

$$\mathtt{P}_s(u) = \{v \in V : \{u, v\} \in E, \mathtt{d}_s(u) = \mathtt{d}_s(v) + 1\}$$

where $\mathtt{d}_s(u)$ and $\mathtt{d}_s(v)$ are the shortest distances from s to u and v, respectively. $\mathtt{P}_s$ defines the *shortest paths graph* rooted in s. Brandes observed that the accumulated dependency values can be computed recursively:

$$\delta_s(v) = \sum_{u : v \in \mathtt{P}_s(u)} \frac{\sigma_{sv}}{\sigma_{su}} \times (1 + \delta_s(u)). \tag{2.3}$$

To compute $\delta_s(v)$ for all $v \in V \backslash \{s\}$, Brandes' algorithm uses a two-phase approach (Algorithm 2). First, a breadth first search (BFS) is initiated from s to compute σ_{sv} and $\mathtt{P}_s(v)$ for each v. Then, in a *back propagation* phase, $\delta_s(v)$ is computed for all $v \in V$ in a bottom-up manner by using (2.3). Each phase considers all the edges at

most once, taking $\mathcal{O}(m)$ time. The phases are repeated for each source vertex. The overall complexity is $\mathcal{O}(mn)$.

Algorithm 2: BC-ORG

```
   Data: G = (V, E)
 1 bc[v] ← 0, ∀v ∈ V
 2 for each s ∈ V do
 3     S ← empty stack, Q ← empty queue
 4     P[v] ← empty list, σ[v] ← 0, dstv ← −1, ∀v ∈ V
 5     Q.push(s),  σ[s] ← 1,  dsts ← 0
       ▷Phase 1: BFS from s
 6     while Q is not empty do
 7         v ← Q.pop(), S.push(v)
 8         for all w ∈ Γ(v) do
 9             if dstw < 0 then
10                 Q.push(w)
11                 dstw ← dstv + 1
12             if dstw = dstv + 1 then
14                 σ[w] ← σ[w] + σ[v]
15                 P[w].push(v)
       ▷Phase 2: Back propagation
16     δ[v] ← 1/σ[v], ∀v ∈ V
17     while S is not empty do
18         w ← S.pop()
19         for v ∈ P[w] do
21             δ[v] ← δ[v] + δ[w]
22         if  w ≠ s then
24             bc[w] ← bc[w] + (δ[w] × σ[w] − 1)
25 return bc
```

2.3 The BADIOS Framework

As mentioned in the introduction, closeness- and betweenness-based graph analysis can be an expensive task. The size of the graph, in particular the size of the largest component in the graph, is the main parameter that affects the practical computation time of many distance-related graph metrics. Hence, compression techniques which can reduce the number of vertices/edges in a graph are promising to make them faster. Furthermore, splitting graphs into multiple connected components, and hence reducing the largest component size, can also help in practice.

BADIOS uses bridges and articulation vertices for splitting graphs. These structures are important since for many vertex pairs s, t, all $s \rightsquigarrow t$ (shortest) paths are passing through them. It also uses three *compression* techniques, based on removing degree-1, side, and identical vertices from the graph. These vertices have special properties: No shortest path is passing through a side-vertex unless the side-vertex is one of the endpoints, all the shortest paths from/to a degree-1 vertex is passing through the same vertex, and for two vertices u and v with identical neighborhoods, $\texttt{bc}[u]$ and $\texttt{bc}[v]$ ($\texttt{cc}[u]$ and $\texttt{cc}[v]$) are equal. A toy graph and a high-level description of the splitting/compression process via **BADIOS** is given in Figure 2.1.

As shown in Figure 2.1, **BADIOS** applies a series of operations as a preprocessing phase: Let $G = G_0$ be the initial graph, and G_ℓ be the one after the ℓth splitting/compression operation. The $\ell + 1$th operation modifies a single connected component of G_ℓ and generates $G_{\ell+1}$. The preprocessing continues if $G_{\ell+1}$ is amenable

to further modification. Otherwise, it terminates and the final CC (or BC) computation begins.

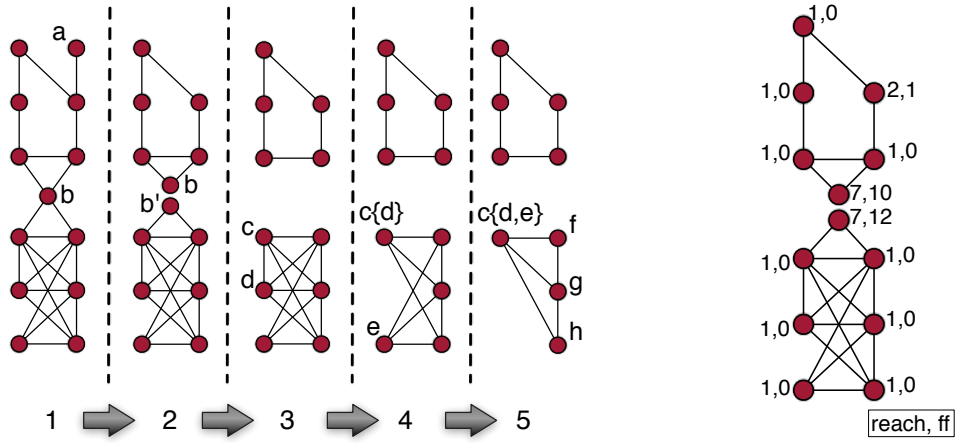


Figure 2.1: (1) a is a degree-1 vertex and b is an articulation vertex. The framework removes a and create a clone b' to represent b in the bottom component. (2) There is no degree-1, articulation, or identical vertex, or a bridge. Vertices b and b' are now side vertices and they are removed. (3) Vertex c and d are now type-II identical vertices: d is removed, and c is kept. (4) Vertex c and e are now type-I identical vertices: e is removed, and c is kept. (5) Vertices c and g are type-II identical vertices and f and h are now type-I identical vertices. The last reductions are not shown but the bottom component is compressed to a singleton vertex. The 5-cycle above cannot be reduced. Rightmost figure shows the situation of reach and ff values in the second stage of manipulation. Values are shown next to each vertex.

Exploiting the existence of above mentioned structures on CC and BC computations can be crucial. For example, all non-leaf vertices in a binary tree $T = (V, E)$ are articulation vertices. When Brandes' algorithm is used, the complexity of BC computation is $\mathcal{O}(n^2)$. One can do much better: Since there is exactly one path between

each vertex pair in V, for $v \in V$, $\texttt{bc}[v]$ is equal to the number of pairs communicating via v, i.e., $\texttt{bc}[v] = 2 \times ((l_v r_v) + (n - l_v - r_v - 1)(l_v + r_v))$ where l_v and r_v are the number of vertices in the left and right subtrees of v, respectively. This approach takes only $\mathcal{O}(n)$ time. These equations can also be modified for closeness-centrality computations and a linear-time CC algorithm can easily be obtained for trees.

A novel feature of **BADIOS** is fully exploiting the above mentioned structures by employing an iterative preprocessing phase. Specifically, a degree-1 removal can create new degree-1, identical, and side vertices. Or, a splitting can reveal new degree-1 and side vertices. Similarly, by removing an identical vertex, new identical, degree-1, articulation, and side vertices can appear. And lastly, new identical and degree-1 vertices can be discovered when a side vertex is removed from the graph. To fully reduce the graph by using the newly formed structures, the framework uses a loop where each iteration performs a set of manipulations on the graph.

2.4 BADIOS for Closeness Centrality

Based on the combinatorial structures mentioned above, we describe a set of *closeness-preserving* graph manipulation techniques to make a graph smaller and disconnected while preserving the information required to compute the distance-based metrics by using some auxiliary arrays. The proposed techniques will especially be useful on expensive distance-based graph kernels such as closeness centrality which will be our main application while describing the proposed approach.

For simplicity, we assume that the graph is initially connected. In order to correctly compute the shortest-path distances and closeness centrality values after reduction, we keep a representative vertex id for some of the vertices removed from the graph during the process. We also assign two auxiliary attributes to all the vertices: `reach` and `ff` (*forwardable farness*).

As explained above, **BADIOS** compresses the graph G, splits it into multiple disconnected components, and obtains another graph $G' = (V', E')$ with several graph manipulations. Let u be a vertex in V' and C' be the connected component of G' containing u. Let $\mathcal{R}_u$ be the set of vertices $v \in (V \setminus C') \cup \{u\}$ such that all the shortest $v \rightsquigarrow w$ paths in the original graph G are passing through u for all $w \in C'$. In G', all the vertices $\mathcal{R}_u \setminus \{u\}$ are disconnected from the vertices in C'. Hence, for each vertex $v \in \mathcal{R}_u$, u will act as a *representative* (or proxy) in C'. During the CC computation, it will be responsible to propagate the impact of v to the closeness centrality values of all the vertices in C'. We use $\texttt{reach}[u] = |\mathcal{R}_u|$ to denote the number of vertices represented by u.

In addition to `reach`, we assign another attribute `ff` to each vertex where at any time of the graph manipulation process

$$\texttt{ff}[u] = \sum_{v \in \mathcal{R}_u} \texttt{dst}_G(u, v).$$

The correctness of the proposed approach heavily depends on the correctness of the updates on these attributes during the process. Before the manipulations, $\texttt{reach}[u]$

is set to 1 for each $u \in V$ since there is only one vertex (itself) in $\mathcal{R}_u$. Similarly, $\texttt{ff}[u]$ is set to 0 since $\texttt{dst}_G(u, u) = 0$.

2.4.1 Closeness-preserving graph splits

We used two approaches to split the graphs into multiple disconnected components; *articulation vertex cloning* and *bridge removal*. Indeed, a bridge exists only between two articulation vertices but we still handle it separately, since we observed that a bridge removal is cheaper and more effective than articulation vertex cloning and the former does not increase the number of vertices but the latter does.

Articulation vertex cloning

Let u be an articulation vertex in a component C appeared in the preprocessing phase where we perform graph manipulations. We split C into k components C_i for $1 \leq i \leq k$ by removing u from G and adding a *local clone* u'_i of u to each new component C_i by connecting u'_i to the same vertices u was connected in C_i as shown in Figure 2.2. For CC computations, to keep the relation between the clones and the original vertex, we use a mapping **org** from V' to V where $\textbf{org}(u'_i)$ is original vertex $u \in V$ for a clone $u'_i \in V$. At any time of a CC preprocessing phase, a vertex $u \in V$ has exactly one *representative* u' in each component C such that $\texttt{reach}[u']$ is increased due to the existence of u. This vertex is denoted as $\textbf{rep}(C, u)$. Note that each local clone is a representative of its original.

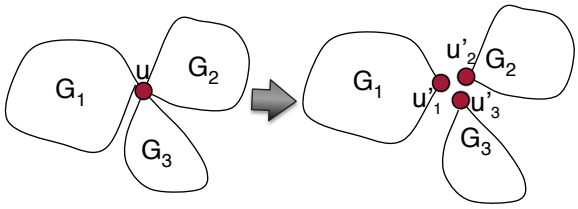


Figure 2.2: Articulation vertex cloning on a toy graph with three disconnected components after the graph manipulation.

The cloning operation keeps the number of edges constant but increases the number of vertices in the graph. The `reach` value for each clone u'_i is set to

$$\texttt{reach}[u'_i] = \texttt{reach}[u] + \sum_{v \in C \setminus C_i} \texttt{reach}[v] \tag{2.4}$$

and its forwardable farness is set to

$$\texttt{ff}[u'_i] = \texttt{ff}[u] + \sum_{\substack{1 \le j \le k \\ j \ne i}} \sum_{v \in C_j} \texttt{dst}_{C_j}(u'_j, v) \tag{2.5}$$

for $1 \le i \le k$. Note that these updates are only local to clone vertices, i.e., only their `reach` and `ff` values are affected. For example, a clone vertex u'_i sees the impact of the $\texttt{dst}_C(u, v)$ on $\texttt{ff}[u'_i]$ even though $v \in C_j$, $i \ne j$, is in another component after the split. However, the same is not true for a non-clone vertex $w \notin C_j$. Hence, considering v and w are not connected anymore, the original CC kernel in Algorithm 1 will not compute the correct closeness centrality values. To alleviate this, we will modify the original kernel later to propagate the forwardable farness values of the clone vertices

to their components. With the modified kernel, we will have

$$\texttt{cc}[u] = \texttt{cc}'[u'_i] \tag{2.6}$$

for $1 \leq i \leq k$. That is, all the vertices cloned from the same articulation vertex will have the same CC after the execution of the modified kernel. Furthermore, this value will be equal to actual centrality of the articulation vertex used for splitting.

Bridge removals

As mentioned above, bridges can only exist between two articulation vertices. The graph can be split into three disconnected components via articulation vertex cloning where one of the components will be a trivial one having a single edge and two clone vertices. Here we show that removal of a bridge $\{u, v\}$ can combine these steps and does not form such unnecessary trivial components. Let C_u and C_v be the two components after bridge removal which contain u and v, respectively. We update the `reach` values of u and v as follows:

$$\texttt{reach}[u] = \texttt{reach}[u] + \sum_{w \in C_v} \texttt{reach}[w], \tag{2.7}$$

$$\texttt{reach}[v] = \texttt{reach}[v] + \sum_{w \in C_u} \texttt{reach}[w]. \tag{2.8}$$

Consecutively, the `ff` values are updated as

$$\texttt{ff}[u] = \texttt{ff}[u] + \left(\texttt{ff}[v] + \sum_{w \in C_v} \texttt{dst}_{C_v}(v, w) \right) + \texttt{reach}[v],$$
$$\texttt{ff}[v] = \texttt{ff}[v] + \left(\texttt{ff}[u] + \sum_{w \in C_u} \texttt{dst}_{C_u}(u, w) \right) + \texttt{reach}[u],$$

where $\texttt{reach}[v]$ and $\texttt{reach}[v]$ are the recently updated values from (2.7) and (2.8). Note that the above equations add the forwardable farness value to each other in addition to the total distance we lose by disconnecting a connected component into two. The last `reach` term is required since $\texttt{reach}[v]$ ($\texttt{reach}[u]$) vertices added to $\mathcal{R}_u$ ($\mathcal{R}_v$), and for all these vertices, v (u) is one edge closer than u (v). Again these values will be propagated to the other vertices in C_u and C_v by the modified CC kernel that will be described later.

To update the `reach` and `ff` values, both the cloning and removal techniques described above require a traversal within the component of the graph in which the articulation vertex or bridge appears. Although it seems costly, the benefit of such manipulations can be understood if the superlinear complexity of CC computation is considered. Assume that a graph is split into k disconnected components each having equal number of vertices and edges. Considering the $\mathcal{O}(n(m+n))$ time complexity, the CC computation for each of these components will take k^2 times less time. Since there are k of them, the split will provide a k fold speedup in total. Although such

articulation vertices and bridges that evenly split the graph do not appear in real-world graphs, even with imbalanced splits, one can obtain significant speedups since the cost of a split is just a single BFS traversal.

2.4.2 Closeness-preserving graph compression

In this section, we present two closeness-preserving techniques which can be used to reduce the number of vertices and edges in a graph: (1) degree-1 vertex removal and (2) side-vertex removal.

Compression with degree-1 vertices

A degree-1 vertex is a special instance of a bridge and can be handled as explained in the previous section. However, the previous approach traverses the entire component once to update the `reach` and `ff` values. Here we propose another approach with $\mathcal{O}(1)$ operations per vertex removal which requires a post-processing after the CC scores of the remaining vertices are computed by the modified kernel.

Figure 2.3 shows a simple example where a degree-1 vertex u appears after the subgraph G_2 is compressed into a single vertex after a set of graph manipulations. To remove u, which is connected to v, three operations need to be performed: (1) an update on `reach`$[v]$, (2) an update on `ff`$[v]$, and (3) setting u as a dependent of v for post-processing. When u is removed, all the vertices that were being represented by u (which are the vertices in G_2) will be represented by v. Hence, the new value of

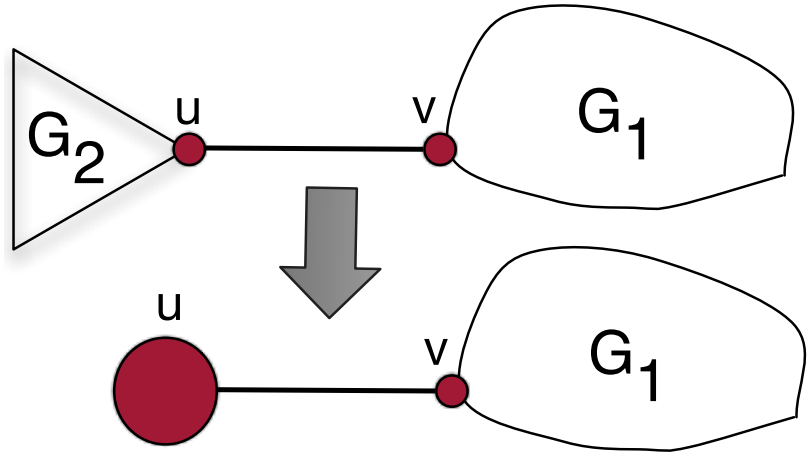


Figure 2.3: A toy graph where G_2 is compressed via manipulations and a degree-1 vertex u is obtained.

$\texttt{reach}[v]$ is updated as

$$\texttt{reach}[v] = \texttt{reach}[v] + \texttt{reach}[u]. \tag{2.9}$$

The forwardable farness of u, i.e., $\texttt{ff}[u]$, needs to be added to $\texttt{ff}[v]$ as

$$\texttt{ff}[v] = \texttt{ff}[v] + \texttt{ff}[u] + \texttt{reach}[u]. \tag{2.10}$$

Similar to the bridge removal case, the last term $\texttt{reach}[u]$ is required in the equation since all the $\texttt{reach}[u]$ vertices that changed their representative to v were one edge closer to u compared to v. As the last operation, we mark that u is dependent to v

and the difference between the overall farness values of u and v is set to

$$\texttt{far}[u] - \texttt{far}[v] = (|V| - \texttt{reach}[u]) - \texttt{reach}[u] \tag{2.11}$$

$$= |V| - 2 \times \texttt{reach}[u]. \tag{2.12}$$

The first term $(|V| - \texttt{reach}[u])$ in the summation is added since all the vertices in V are one edge far away, except the ones in $\mathcal{R}_u$, to u compared to v. Similarly, all the vertices in $\mathcal{R}_u$ are one edge closer to u. Thus we have an additional $-\texttt{reach}[u]$ in (2.11). Sum of these two terms give the dependency equation in (2.12), i.e., the difference in u and v's farness. Hence, once the overall farness value of v is computed, the farness value of u can be computed via a simple addition during a post-processing phase.

Compression with side vertices

Let u be a side vertex appearing in a component during the graph manipulation process. Since $\Gamma(u)$ is a clique, except the ones starting or ending at u, no shortest path is passing through u, i.e., u is always on the sideways. Hence, we can remove u if we compensate the effect of the shortest $s \rightsquigarrow t$ paths where u is either s or t. To do this, we initiate a BFS from u in the original graph G as shown in Algorithm 3.

The main difference between a BFS in side-vertex removal and in the original implementation in Algorithm 1 is line 13 (of Algorithm 3) which adds $\texttt{dst}[w]$ to $\texttt{far}[w]$ for each traversed vertex w. To do that, a single variable to store the farness value (as in Algorithm 1) is not sufficient since side-vertex removals update the farness values

Algorithm 3: Side-vertex removal BFS for closeness centrality

```
   Data: side vertex u, G = (V, E), far[.]
 1 Q ← empty queue
 2 Q.push(u)
 3 dst[u] ← 0
 4 dst[v] ← ∞, ∀v ∈ V \ {u}
 5 while Q is not empty do
 6 |   v ← Q.pop()
 7 |   for all w ∈ Γ_G(v) do
 8 |   |   if dst[w] = ∞ then
 9 |   |   |   Q.push(w)
10 |   |   |   dst[w] ← dst[v] + 1
11 |   |   |   far[u] ← far[u] + dst[w]
13 |   |   |   far[w] ← far[w] + dst[w]
14 cc[u] ← 1/far[u]
```

partially and these updates need to be stored till the end of the graph manipulation process. Hence, we used an additional `far` array to perform side-vertex removal operations.

This compression technique has a little impact of the overall time since for a side vertex removal, an additional BFS (Algorithm 3) is necessary and it is almost as expensive as the original BFS (of Algorithm 1) we try to avoid. However, these removals can make new special vertices appear during the manipulation process which enable further splits and compression of the graph in a cheaper way.

2.4.3 Combining and post-processing

We continuously process a reduction on the graph with split and compression operations until no further reduction possible. We first perform degree-1 removals

since they are the cheapest to handle. Next, we split the graph by first bridges and then articulation vertex clones. The order is important for efficiency since the former is cheaper. We iteratively use these three techniques until no reduction is possible. After that we remove the side vertices to discover new special vertices. The reason behind delaying the side-vertex removals is that its additional BFS requirement makes it expensive compared to the other graph manipulation techniques. Hence, we do not use them until we really need them.

After all the graph manipulation techniques, the original CC kernel given in Algorithm 1 cannot compute the correct centrality values since it does not forward the `ff` values to the other vertices. We apply a modified version as shown in Algorithm 4 to compute the CC scores once the split and compression operations are done and `reach` and `ff` attributes are fixed.

Theorem 1. *Let $G = (V, E)$ be the original graph and $G' = (V', E')$ is the reduced graph after split and compression operations with* `reach`*,* `ff`*, and* `far` *attributes computed for each vertex $v \in V'$. Assuming these attributes are correct, for all the vertices in V', the CC scores of G computed by Algorithm 1 is the same with the CC scores of G' computed by Algorithm 4.*

Proof. For a source vertex $s \in V'$ and another vertex $w \neq s$ that is connected to s in G', $\texttt{ff}[w]$ is forwardable to $\texttt{far}[s]$ by using the equation at lines 10 and 12 of Algorithm 4. Remember that for a vertex $w \in G'$, all the $\texttt{reach}[w]$ vertices in $\mathcal{R}_w$ are not connected to s. Hence, they are represented by w and from s (and from any vertex in the same component), they are reachable only through w. Since the shortest-path distance between s and w is $\texttt{dst}[w]$, the vertices in $\mathcal{R}_w$ are $\texttt{dst}[w]$ more edges far away from s when compared to w. Thus an additional $\texttt{dst}[w] \times \texttt{reach}[w]$ farness is required while forwarding the $\texttt{ff}[w]$ value to $\texttt{far}[s]$.

At the end of the algorithm (line 14), we have an extra addition of $\texttt{ff}[s]$ to the total farness value of s. It is required since while computing the total farness of s and its `cc` score, we need to consider the farness due to the vertices in $\mathcal{R}_s$. □

Algorithm 4: CC-REACH: Modified closeness centrality computation

Data: $G' = (V', E')$, `ff`[.], `reach`[.], `far`[.]
Output: `cc`[.]

```
1  for each s ∈ V' do
2      ··· ▷same as CC
3      while Q is not empty do
4          v ← Q.pop()
5          for all w ∈ Γ_{G'}(v) do
6              if dst[w] = ∞ then
7                  Q.push(w)
8                  dst[w] ← dst[v] + 1
10                 fwd ← ff[w] + (dst[w] × reach[w])
12                 far[s] ← far[s] + fwd
14     far[s] ← far[s] + ff[s]
15     cc[s] ← 1/far[s]
16 return cc[.]
```

Work filtering with identical vertices

If some vertices in G' are identical, i.e., their adjacency lists are the same, the forwardable farness values from other vertices to their overall farness will be the same. Hence, it is possible to combine these vertices and avoid extra computation in Algorithm 4. We use 2 types of identical vertices: Vertices u and v are type-I (or type-II) identical if and only if $\Gamma(u) = \Gamma(v)$ (or $\Gamma(u) \cup \{u\} = \Gamma(v) \cup \{v\}$), as exemplified in Figure 2.4.

The compression works as follows: Let $G' = (V', E')$ be the reduced graph after preprocessing operations, and let $\mathcal{I} \subset V'$ be a set of identical vertices. We select a proxy vertex $u \in \mathcal{I}$, compute its overall farness to other vertices and CC score as

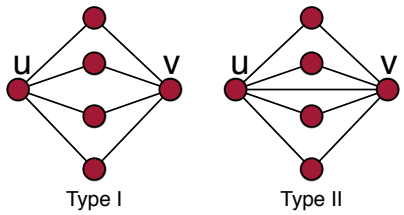


Figure 2.4: Type-I (left) and type-II (right) identical vertices u and v.

shown in Algorithm 4. Then for a vertex $v \in \mathcal{I}$, we compute

$$\texttt{far}[v] = \texttt{far}[u] - k \times (\texttt{reach}[v] - \texttt{reach}[u]), \tag{2.13}$$

$$\texttt{cc}[v] = 1/\texttt{far}[v], \tag{2.14}$$

where k, the shortest distance between two identical vertices, is 2 for a type-I identical vertex set, and 1 for a type-II identical vertex set. Note that the only difference between the farness values is $k \times (\texttt{reach}[v] - \texttt{reach}[u])$ according to the lines 10, 12, and 14.

Post-processing for the degree-1 vertices

Once Algorithm 4 is done, the only remaining part is computing the CC scores of removed degree-1 vertices since they are not in G' anymore. To do that, we resolve the dependencies created when the degree-1 vertices are being removed. We do a loop on the vertices and for each vertex u we visit, we check if u's CC score is already computed. If not, we recursively follow the dependencies to find the final representative

vertex in G'. While coming back from the recursion path, we use Equation (2.12) to find the farness and the CC score(s) of the removed degree-1 vertices. Since the dependencies form a tree and at most $\mathcal{O}(1)$ operations are performed per vertex, we need at most $\mathcal{O}(|V|)$ operations to resolve all the dependencies.

2.5 BADIOS for Betweenness Centrality

Here we propose a set of *betweenness-preserving* graph manipulation techniques similar to the ones described for closeness centrality. The proposed techniques will make the original graph $G = (V, E)$ smaller and disconnected while preserving the information required to compute the distance-based metrics by using some auxiliary arrays.

2.5.1 Betweenness-preserving graph splits

To correctly compute the BC scores after splitting G, we use the `reach` attribute as described above and set $\texttt{reach}[v] = 1$ for all $v \in V$ before the manipulations.

Articulation vertex cloning

Let u be an articulation vertex in a component C obtained during the preprocessing phase whose removal splits C into k (connected) components C_i for $1 \leq i \leq k$. As in CC, we remove u and keep a local clone u'_i at each component C_i. For betweenness centrality on **BADIOS**, the `reach` values for each local clone is set with

$$\texttt{reach}[u'_i] = \sum_{v \in C \setminus C_i} \texttt{reach}[v] \tag{2.15}$$

for $1 \leq i \leq k$.

Algorithm 5: BC-REACH: Modified betweenness centrality computation

```
   Data: G' = (V', E') and reach
 1 bc'[v] ← 0, ∀v ∈ V'
 2 for each s ∈ V' do
 3 |   ··· ▷same as BC-ORG
 4 |   while Q is not empty do
 5 |   |   ··· ▷same as BC-ORG
 7 |   δ[v] ← reach[v] − 1, ∀v ∈ V'
 8 |   while S is not empty do
 9 |   |   w ← S.pop()
10 |   |   for v ∈ P[w] do
12 |   |   |   δ[v] ← δ[v] + (σ[v]/σ[w]) × (1 + δ[w])
13 |   |   if w ≠ s then
15 |   |   |   bc'[w] ← bc'[w] + (reach[s] × δ[w])
16 return bc'
```

Algorithm 5 computes the BC scores of the vertices in a split graph. Note that the only difference with BC-ORG are lines 7 and 15, and if $\texttt{reach}[v] = 1$ for all $v \in V$, then the algorithms are equivalent. Hence, the complexity of BC-REACH is also $\mathcal{O}(mn)$ for a graph with n vertices and m edges.

Let $G = (V, E)$ be the initial graph, $|V| = n$, and $G' = (V', E')$ be the split graph obtained via preprocessing. Let $\texttt{bc}$ and $\texttt{bc}'$ be the scores computed by BC-ORG(G) and BC-REACH(G'), respectively. We will prove that

$$\texttt{bc}[v] = \sum_{v' \in V' | \mathbf{org}(v') = v} \texttt{bc}'[v'], \tag{2.16}$$

when the graph is split at articulation vertices. That is, $\texttt{bc}[v]$ is distributed to $\texttt{bc}'[v']$s where v' is a local clone of v. Let us start with two lemmas.

Lemma 1. *Let u, v, s be vertices of G such that all $s \rightsquigarrow v$ paths contain u. Then, $\delta_s(v) = \delta_u(v)$.*

Proof. For any target vertex t, if $\sigma_{st}(v)$ is positive then

$$\delta_{st}(v) = \frac{\sigma_{st}(v)}{\sigma_{st}} = \frac{\sigma_{su}\sigma_{ut}(v)}{\sigma_{su}\sigma_{ut}} = \frac{\sigma_{ut}(v)}{\sigma_{ut}} = \delta_{ut}(v)$$

since all $s \rightsquigarrow t$ paths are passing through u. According to (2.2), $\delta_s(v) = \delta_u(v)$. □

Lemma 2. *For any vertex pair $s, t \in V$, there exists exactly one component C of G' which contains a clone of t and a representative of s as two distinct vertices.*

Proof. (*by induction on the number of splits*) Given $s, t \in V$, the statement is true for the initial (connected) graph G since it contains one clone of each vertex. Assume that it is also true after the ℓ-th splitting. Let C be this component. When C is further split via t's clone, all but one newly formed (sub)components contains a clone of t as the representative of s. For the remaining component C', $\mathbf{rep}(C', s) = \mathbf{rep}(C, s)$ which is not a clone of t.

For all components other than C, which contain a clone t' of t, the representative of s is t' by the inductive assumption. When such components are further split, the representative of s will be again a clone of t. Hence the statement is true for $G_{\ell+1}$, and by induction, also for G'. □

The local clones of an articulation vertex v, created while splitting, are acting as the original vertex v in their components. Once the `reach` value for each clone is set as in (2.15), line 7 of Bc-Reach handles the BC contributions from each new component (except the one containing the source), and line 15 of Bc-Reach fixes the contribution of vertices reachable only via the source s.

Theorem 2. *Eq. 2.16 is correct after splitting G with articulation vertices.*

Proof. Let C be a component of G', s', v' be two vertices in C, and s, v be their original vertices in V, respectively. Note that $\texttt{reach}[v'] - 1$ is the number of vertices

$t \neq v$ such that t does not have a clone in C and v lies on all $s \rightsquigarrow t$ paths in G. For all such vertices, $\delta_{st}(v) = 1$, and the total dependency of v' to all such t is $\texttt{reach}[v'] - 1$. When the BFS is started from s', line 7 of BC-REACH initiates $\delta[v']$ with this value and computes the final $\delta[v'] = \delta_{s'}(v')$. This is the same dependency $\delta_s(v)$ computed by BC-ORG.

Let C be a component of G', u' and v' be two vertices in C, and $u = \mathbf{org}(u')$, $v = \mathbf{org}(v')$. According to the above paragraph, $\delta_u(v) = \delta_{u'}(v')$ where $\delta_u(v)$ and $\delta_{u'}(v')$ are the dependencies computed by BC-ORG and BC-REACH, respectively. Let $s \in V$ be a vertex, s.t. $\mathbf{rep}(C, s) = u'$. According to Lemma 1, $\delta_s(v) = \delta_u(v) = \delta_{u'}(v')$. Since there are $\texttt{reach}[u']$ vertices represented by u' in C, the contribution of the BFS from u' to the BC score of v' is $\texttt{reach}[u'] \times \delta_{u'}(v')$ as shown in line 15 of BC-REACH. Furthermore, according to Lemma 2, $\delta_{s'}(v')$ will be added to exactly one clone v' of v. Hence, (2.16) is correct. □

Bridge removals

Let $\{u, v\}$ be a bridge in a component C formed during graph manipulations. Let $u' = \mathbf{org}(u)$ and $v' = \mathbf{org}(v)$. As stated above, a bridge removal operation is similar to a splitting via an articulation vertex, however, no new clones of u' or v' are created. Instead, we let u and v act as a clone of v' and u' in the newly created components C_u and C_v which contain u and v, respectively. Similar to (2.15), we add $\sum_{w \in C_v} \texttt{reach}[w]$ and $\sum_{w \in C_u} \texttt{reach}[w]$ to $\texttt{reach}[u]$ and $\texttt{reach}[v]$, respectively, to make u (v) the representative of all the vertices in C_v (C_u).

After a bridge removal, updating the `reach` values is not sufficient to make Lemma 2 correct. No component contains a distinct representative of u' (v') and clone of v' (u') anymore. Hence, $\delta_v(u')$ and $\delta_u(v')$ will not be added to any clone of

u' and v', respectively, by Bc-Reach. But we can compute the difference and add

$$\delta_v(u) = \left(\left(\sum_{w \in C_u} \texttt{reach}[w]\right) - 1\right) \times \sum_{w \in C_v} \texttt{reach}[w],$$

to $\texttt{bc}'[u]$ and add $\delta_u(v)$ to $\texttt{bc}'[v]$, where $\delta_u(v)$ is computed by interchanging u and v in the right side of the above equation. Note that Lemma 2 is correct for all other vertex pairs.

Corollary 1. *Eq. 2.16 is correct after splitting G with articulation vertices and bridges.*

2.5.2 Betweenness-preserving graph compression

Here we present **BADIOS**'s betweenness-preserving compression techniques: (1) degree-1 vertex removal, (2) compression by identical vertices, and (3) side-vertex removal.

Compression with degree-1 vertices

As stated before, although a degree-1 vertex removal is a special instance of a graph split with a bridge, we handle them separately to avoid trivial components. Let u be a degree-1 vertex connected to v and appeared in a component C formed during the preprocessing. To remove u, we add $\texttt{reach}[u]$ to $\texttt{reach}[v]$ and increase

$\texttt{bc}'[u]$ and $\texttt{bc}'[v]$, respectively, with

$$\delta_v(u) = (\texttt{reach}[u] - 1) \times \sum_{w \in C \setminus \{u\}} \texttt{reach}[w],$$

$$\delta_u(v) = \left(\left(\sum_{w \in C \setminus \{u\}} \texttt{reach}[w] \right) - 1 \right) \times \texttt{reach}[u].$$

Corollary 2. *Eq. 2.16 is correct after splitting G with articulation vertices and bridges, and compressing it with degree-1 vertices.*

Compression with identical vertices

Instead of basic work filtering applied for CC, **BADIOS** uses the type-I and type-II identical vertices to compress the graph further for BC. Hence, it exploits these vertices in a more complex way. To handle the complexity, an `ident` attribute is assigned to each vertex where $\texttt{ident}(v)$ denotes the number of vertices in G that are identical to v in G'. Initially, $\texttt{ident}[v]$ is set to 1 for all $v \in V$.

Let $\mathcal{I}$ be a set of identical vertices formed during the preprocessing phase. We remove all vertices in $\mathcal{I}$ except one, which acts as a proxy for the others. Let v be the proxy vertex for $\mathcal{I}$. We increase $\texttt{ident}[v]$ by $\sum_{v' \in \mathcal{I}, v' \neq v} \texttt{ident}[v']$, and associate a list $\mathcal{I} \setminus \{v\}$ with v. The integration of the identical-vertex compression is realized in three modifications on Algorithm 2: During the first phase, line 14 is changed to $\sigma[w] \leftarrow \sigma[w] + \sigma[v] \times \texttt{ident}[v]$, since v can be a proxy for some vertices other than itself. Similarly, w can be a proxy, and line 21 is modified as $\delta[v] \leftarrow \delta[v] + \frac{\sigma[v]}{\sigma[w]} \times (\delta[w] + 1) \times \texttt{ident}[w]$ to correctly simulate w's identical vertices. Finally, the source s can be a proxy, and the current BFS phase can be a representative

for $\texttt{ident}[s]$ phases. To handle that, the BC updates at line 24 are changed to $\texttt{bc}'[w] \leftarrow \texttt{bc}'[w] + \texttt{ident}[s] \times \delta[w]$. The BC scores of all the vertices in $\mathcal{I}$ are equal.

The only paths ignored via these modifications are the paths between $u \in \mathcal{I}$ and $v \in \mathcal{I}$. If $\mathcal{I}$ is type-II the $u \rightsquigarrow v$ path contains a single edge and has no effect on dependency (and BC) values. However, if $\mathcal{I}$ is type-I, such paths have some impact. Fortunately, it only impacts the immediate neighbors' BC scores of $\mathcal{I}$. Since there are exactly $\sum_{u \in \mathcal{I}}(\texttt{ident}[u](\sum_{v \in \mathcal{I}, u \neq v} \texttt{ident}[v]))$ such paths, this amount is equally distributed among the immediate neighbors of $\mathcal{I}$.

The technique presented in this section has been presented without taking the `reach` attribute into account. Both attributes can be maintained simultaneously. The details are not presented here for brevity. The main challenge is to keep track of the BC of each identical vertex since they can differ if the reach value of the identical vertices are not equal to 1.

Corollary 3. *Eq. 2.16 is correct after splitting G with articulation vertices and bridges, and compressing it with degree-1, and identical vertices.*

Compression with side vertices

Let u be a side vertex in a component C formed after a set of manipulations on the original graph G. Since $\Gamma(u)$ is a clique, no shortest path is passing through u. Hence, we can remove u from C by compensating the effect of the shortest $s \rightsquigarrow t$ paths where u is either s or t. To do this, we initiate a BFS from u similar to the one in Bc-Reach. As Algorithm 6 shows, the only differences are two additional lines 12 and 14. Note that this extra BFS is as expensive as the original one we

avoid by removing u. As in CC, **BADIOS** performs the side-vertex removals since they can yield new special vertices in the graph, which will be used to improve the performance.

Algorithm 6: Side-vertex removal BFS for betweenness centrality

```
   Data: G_ℓ = (V_ℓ, E_ℓ), a side vertex s, reach, and bc'
 1 ··· ▷same as Bc-Reach
 2 while Q is not empty do
 3 |   ··· ▷same as the BFS in Bc-Reach
 4 δ[v] ← reach[v] − 1, ∀v ∈ V_ℓ
 5 while S is not empty do
 6 |   w ← S.pop()
 7 |   for v ∈ P[w] do
 8 |   |   δ[v] ← δ[v] + (σ[v]/σ[w])(1 + δ[w])
 9 |   if w ≠ s then
10 |   |   bc'[w] ← bc'[w] + (reach[s] × δ[w])+
12 |   |                 (reach[s] × (δ[w] − (reach[w] − 1))
14 bc'[s] ← bc'[s] + (reach[s] − 1) × δ[s]
15 return bc'
```

Let v, w be two vertices in C different than u. Although both vertices will keep existing in $C - u$, since u will be removed, $\delta_v(w)$ will be $\texttt{reach}[u] \times \delta_{vu}(w)$ less than it should be. For all such v, the aggregated dependency will be

$$\sum_{v \in C, v \neq w} \delta_{vu}(w) = \delta_u(w) - (\texttt{reach}[w] - 1),$$

since none of the $\texttt{reach}[w] - 1$ vertices represented by w lies on a $v \rightsquigarrow u$ path and $\delta_{vu}(w) = \delta_{uv}(w)$. The same dependency appears for all vertices represented by u. Line 12 of Algorithm 6 takes into account all these dependencies.

Let $s \in V$ be a vertex s.t. $\mathbf{rep}(C, s) = v \neq u$. When we remove u from C, due to Lemma 2, $\delta_s(u) = \delta_v(u)$ will not be added to any clone of $\mathbf{org}(u)$. Since, u is a side vertex, $\delta_v(u) = \texttt{reach}[u] - 1$. Since there are $\sum_{v \in C-u} \texttt{reach}[v]$ vertices which are represented by a vertex in $C - u$, we add

$$(\texttt{reach}[u] - 1) \times \sum_{v \in C-u} \texttt{reach}[v]$$

to $\texttt{bc}'[u]$ after removing u from C. Line 14 of Algorithm 6 compensates this loss.

Corollary 4. *Eq. 2.16 is correct after splitting G with articulation vertices and bridges, and compressing it with degree-1, identical, and side vertices.*

2.5.3 Combining the techniques

For betweenness centrality, **BADIOS** first applies degree-1 removal since it is the cheapest to handle. Next, it splits the graph by first removing the bridges, and then articulation vertices. It then removes the identical vertices in the graph in the order of type-II and type-I. Notice that type-II removals can reveal new type-I identical vertices but the reverse is not possible. The framework iteratively uses these 4 techniques until it reaches a point where no reduction is possible. At that point, it removes the side vertices to discover new special vertices. Similar to CC, the framework does not use side vertices until it really needs them.

2.6 Experiments

We implemented our framework in `C++`. The code is compiled with `gcc v4.8.1` and optimization flag `-O2`. The graph is kept in memory in the Compressed Storage by

Row format (essentially adjacency list which is compact in memory). The experiments are run on a computer with Intel Xeon E5520 CPU clocked at 2.27GHz and equipped with 48GB of main memory. All the experiments are run sequentially.

For the experiments, we used 13 real-world networks from the UFL Sparse Matrix Collection (http://www.cise.ufl.edu/research/sparse/matrices/). Their properties are summarized in Table 2.1. They are from different application areas, such as grid (*power*), router (*as-22july06*, *p2p-Gnutella31*), social (*PGPgiantcompo*, *astro-ph*, *cond-mat-2005*, *soc-sign-epinions*, *loc-gowalla*, *amazon0601*, *wiki-Talk*), protein interaction (*protein*), and web networks (*web-NotreDame*, *web-Google*). We symmetrized the directed graphs. We categorized the graphs into two classes; small and large ones (separately shown in Table 2.1).

Our proposed techniques can be combined in many different ways. In this section we use lower case abbreviations for representing these combined methods. We will use lower case letters *'o'* for the BFS *o*rdering, *'d'* for *d*egree-1 vertices, *'b'* for *b*ridge, *'a'* for *a*rticulation vertices, *'i'* for *i*dentical vertices, and *'s'* for *s*ide vertices. The ordering is performed to improve the cache locality during centrality computation by initiating a BFS from a random source vertex as in Algorithm 1 and renumbering the vertices as their visit order. Using this scheme, for example, abbreviation *das* means that the degree-1 removal is followed by the articulation vertex cloning, which is followed by the side-vertex removal. This pattern is repeated until no further modification is possible.

Graph			Time (in sec.)					
name	$\|V\|$	$\|E\|$	BC org.	BC best	BC Sp.	CC org.	CC best	CC Sp.
as-22july06	22.9K	48.4K	43.72	8.78	4.9	17.03	5.49	3.1
astro-ph	16.7K	121.2K	40.56	19.41	2.0	14.10	9.15	1.5
cond-mat-2005	40.4K	175.6K	217.41	97.67	2.2	79.16	46.21	1.7
p2p-Gnutella31	62.5K	147.8K	422.09	188.14	2.2	180.27	65.13	2.8
PGPgiantcompo	10.6K	24.3K	10.99	1.55	7.0	4.63	0.75	6.2
power	4.9K	6.5K	1.47	0.60	2.4	0.78	0.27	2.8
protein	9.6K	37.0K	11.76	7.33	1.6	4.12	2.33	1.7
			geometric mean		**2.8**	**geometric mean**		**2.5**
amazon0601	403K	2,443K	42,656	36,736	1.1	17,653	11,901	1.5
loc-gowalla	196K	950K	5,926	3,692	1.6	2,117	1,138	1.9
soc-sign-epinions	131K	711K	2,193	839	2.6	889	264	3.4
web-Google	875K	4,322K	153,274	27,581	5.5	83,821	22,935	3.7
web-NotreDame	325K	1,090K	7,365	965	7.6	2,736	517	5.3
wiki-Talk	2,394K	4,659K	452,443	56,778	7.9	279,548	22,029	12.7
			geometric mean		**3.4**	**geometric mean**		**3.7**

Table 2.1: The graphs used in the experiments. Columns *BC org.* and *CC org.* show the original execution times of BC and CC computation without any modification. And *BC best* and *CC best* are the minimum execution times achievable via our framework for BC and CC. The names of the graphs are kept short where the full names can be found in the text.

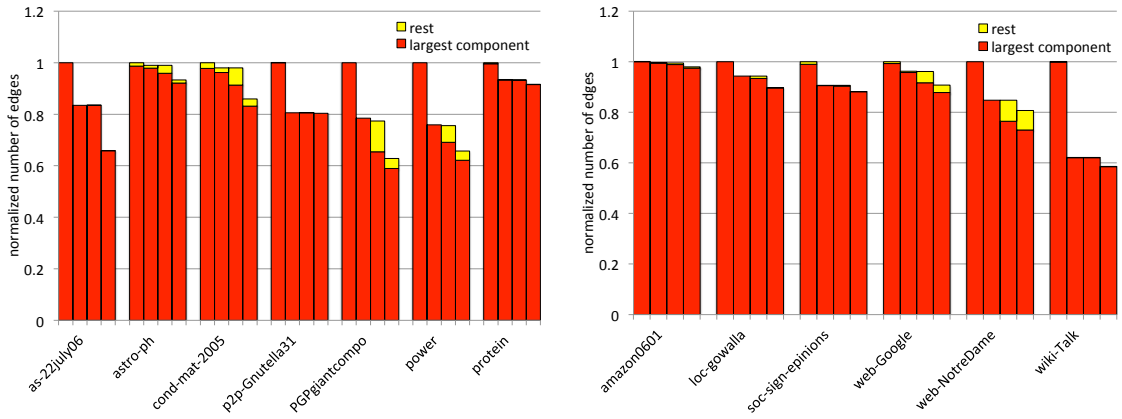
2.6.1 Closeness centrality experiments

We first investigate the efficiency of **BADIOS** on reducing the graphs. We check the number of remaining edges by applying our techniques on the test graphs. Figures 2.5(a) and 2.5(b) show the number of remaining edges in the reduced graph normalized with respect to the original number of edges in G. We chose the variants, *d*, *da*, and *das* since these manipulations are the only ones that reduce the number of edges or make new articulation vertices appear. We measured the remaining number of edges in the largest connected component as well as the other components (shown as "rest"). Degree-1 vertex removal (going from 1st bar to 2nd bar) provides

13% and 14% average reductions in the sizes of small and large graphs, respectively. This result shows that there is a significant amount of degree-1 vertices in real-world graphs and they can be efficiently utilized by our techniques. When we measure the impact of articulation vertex cloning on total number of connected components, we observe two facts: (1) there is usually one giant (strongly) connected component in real-world social networks, and (2) other components are small in size. As can be seen from the 2nd and 3rd bars, articulation-vertex cloning increases the yellow colored regions in the graph, i.e., splits the graphs. Lastly, we measure the effect of side vertex removal. The differences between the 3rd and 4th bars show the reduction by side vertex removal. We observe 9% and 5% average reductions in small and large graphs.

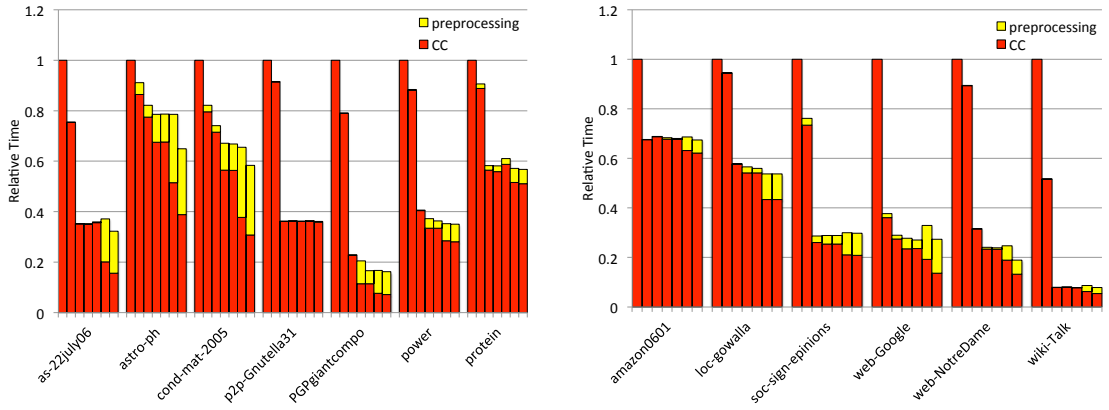
Next, we measure the performance of **BADIOS** on CC computation time. We evaluate the preprocessing and computation time separately. Figures 2.6(a) and 2.6(b) present the runtimes for each combination normalized w.r.t. the implementation of Algorithm 1. For each graph, we tested 6 different combinations of the improvements proposed in this work: They are denoted with *o*, *do*, *dao*, *dbao*, *dbaos*, and *dbaosi*. For each graph, each figure has 7 stacked bars for the 6 combinations in the order described above plus the base implementation.

In many graph kernels, the order of edge accesses is important to due to cache locality. Therefore, we order our graphs after split and compression operations. The second bars for each graph at Figures 2.6(a) and 2.6(b) show the improvement gained



(a) Normalized remaining edges for small graphs (b) Normalized remaining edges for large graphs

Figure 2.5: The plots on the left and right show the number of remaining edges on the graphs which initially have less than and more than 500K edges, respectively. They show the ratio of remaining edges of the variants, which consecutively reduce the number of edges: base, *d*, *da*, *das*. The number of remaining edges are normalized w.r.t. total number of edges in the graph and divided into two: largest connected component and rest of the graph.



(a) Normalized execution times for small graphs (b) Normalized execution times for large graphs

Figure 2.6: The plots on the left and right show the CC computation times on graphs with less than and more than 500K edges, respectively. They show the normalized runtime of the variants: base, *o*, *do*, *dao*, *dbao*, *dbaos*, *dbaosi*. The times are normalized w.r.t. base and divided into two: preprocessing, and the CC computation.

by ordering the graphs. We have 13% and 34% improvements (over the baseline) with ordering for small and big graphs, respectively. Especially larger graphs benefit more from the graph ordering and the cache is utilized more efficiently.

In general, the preprocessing phase takes little time for all graphs. At most 7% of the overall execution time is spent for graph manipulations on small graphs and this value is 6% for large graphs. With split and compression operations, **BADIOS** can obtain significant speedup values. When we only remove the degree-1 vertices, we have 16% runtime improvement for small graphs and 54% improvement for large graphs. When Figures 2.5(a) and 2.5(b), are compared with Figures 2.6(a) and 2.6(b),

the correlation between the reduction on the number of edges and the improvement on the performance becomes more clear. In addition to degree-1 removal, if we split the input graph with articulation vertex cloning, the speedups increase: in large graphs, this reduces the overall execution time up to 5%. As expected, when there are more articulation vertices in the graph, the speedups are higher. As explained in Section 2.4.1, a bridge always exists between two articulation vertices but bridge removal is cheaper than articulation vertex cloning. We see the effect of cheap bridge removals when we look at the combination *odab* (5th bar): in small graphs, we have 4% improvement with articulation vertex cloning plus bridge removal over only articulation vertex cloning.

The side vertex removals turn out to be not efficient. We can not observe significant speedups when we remove the side vertices in graphs. On the other hand, filtering the work via identical vertices brings good improvements. We gain 8% and 10% in small and large graphs with identical vertex filtering. This shows that there are significant amount of identical vertices in the reduced graph and they can be utilized for faster solutions.

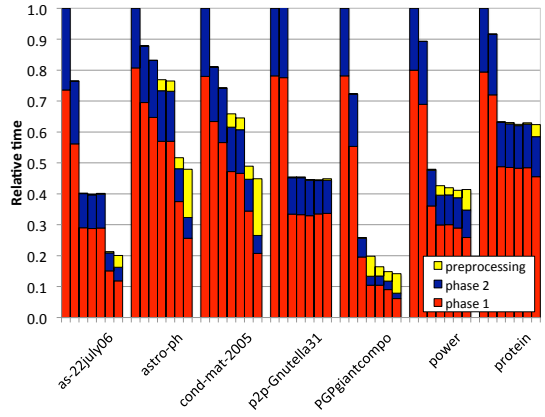
Overall, we have decent speedup numbers for CC when all the techniques are applied. Table 2.1 shows the runtime of the base algorithm, runtime of the combination where all techniques are used, and the speedup obtained by that combination. For the largest graph we have, *wiki-Talk* with 2.3M vertices and 4.6M edges, we reach a speedup of 12.7 over the base implementation.

2.6.2 Betweenness centrality experiments

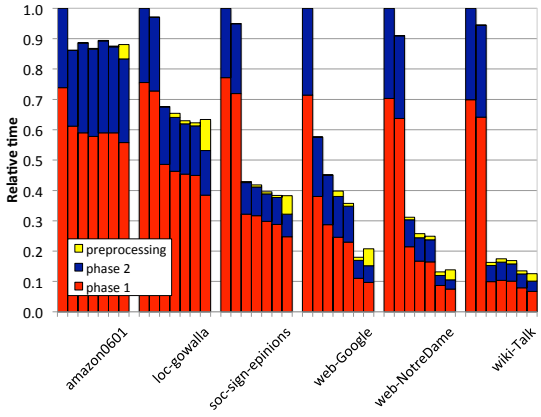
Here we experimentally evaluate the performance of **BADIOS** for betweenness centrality computations. As we did for CC, we measure the preprocessing time and BC computation time separately. Figures 2.7(a) and 2.7(b) present the runtimes for each combination normalized w.r.t. Brandes' algorithm. For each graph, each figure has 7 stacked bars for the 7 combinations in the order described in the caption. To compare the reductions on the execution times with the reductions on the number of edges and vertices, in Figures 2.7(c)–2.7(d), the number of edges remaining in the graph after the preprocessing phase are given for the combinations *d*, *da*, *dai*, and *dasi*.

As Figure 2.7 shows, there is a direct correlation between the amount of edges remaining after the graph manipulations and the overall execution time (except for *soc-sign-epinions* and *loc-gowalla* with 12% and 11% decrease in number of vertices, respectively). This proves that our rationale behind investigating splitting and compression techniques is valid also for BC.

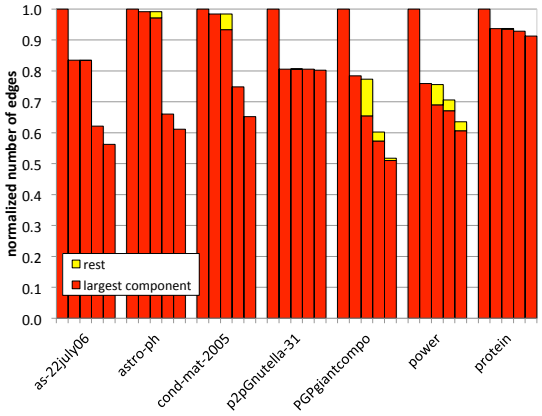
Table 2.1 shows the runtime of the base BC algorithm as well as the runtime of the combination that lead to the best improvement and the speedup obtained by that combination. Almost for all graphs, **BADIOS** provides a significant improvement. We observe up to 7.9 speedup on large graphs. For *wiki-Talk*, applying all techniques reduced the runtime from 5 days to 16 hours.



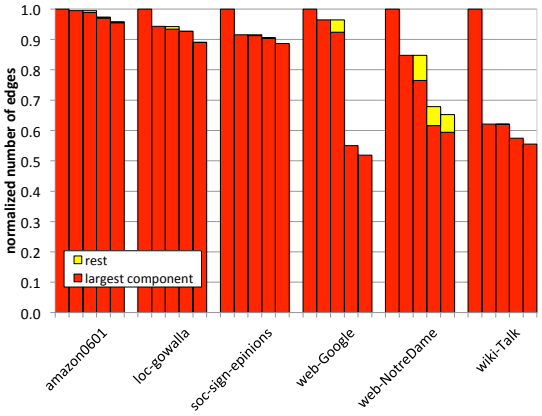
(a) Normalized execution times for small graphs



(b) Normalized execution times for large graphs



(c) #remaining edges for small graphs



(d) #remaining edges for large graphs

Figure 2.7: The plots on the left and right show the results on graphs with less than and more than 500K edges, respectively. The top plots show the runtime of the variants: base, *o*, *do*, *dao*, *dbao*, *dbaio*, *dbaiso*. The times are normalized w.r.t. base and divided into three: preprocessing, the first phase and the second phase of the BC computation. The bottom plots show the number of edges in the largest 200 components after preprocessing.

Although it is not that common, applying degree-1- and identical-vertex removal can degrade the performance by a small amount. When the number of vertices removed is small, their removal does not compensate the overhead induced by the `reach` and `ident` attributes in the algorithms. The only graph **BADIOS** does not perform well on is the co-purchasing network of Amazon website, *amazon0601*, where it brings less than 20% of improvement. This graph contains large cliques formed by the users purchasing the same item, and hence does not have enough number of special vertices.

2.7 Related Work

Several techniques have been proposed to cope with large networks with limited success either by using approximate computations [31, 69], or by throwing hardware resources to the problem by parallelizing the computations on distributed memory architectures [108], multicore CPUs [116], and GPUs [164, 90].

To the best of our knowledge, there are two concurrent works since our first release, noted in our technical report [152]. However, their focus is limited to BC computation only. The first work introduces degree-1 vertex removal for BC [17]. In the second, Puzis et al. propose to remove articulation vertices and structurally equivalent vertices which correspond to our type-I identical vertices [137]. We did not compare our speedups with theirs for three reasons: the techniques they use form only a subset of the techniques we proposed in this work, they are not well integrated as we did in **BADIOS**, and even our base implementation is already 40–45 times

faster than their fastest algorithm (see the results for *soc-sign-epinions* [17] and *p2p-Gnutella31* [137]). We believe that an efficient implementation of a novel algorithm is mandatory to evaluate any improvement.

2.8 Summary

In this work, we proposed the **BADIOS** framework to reduce the execution time of betweenness and closeness centrality computations. It uses techniques that break graphs into pieces while keeping the information to recompute the pair and source dependencies, which are the building blocks of BC scores, and the information to preserve closeness values, for CC. It also uses some compression techniques to reduce the number of vertices and edges. Combining these techniques provides great reductions in graph sizes and component numbers. An experimental evaluation with various networks shows that the proposed techniques are highly effective in practice and they can be a great arsenal to reduce the execution time for BC computation. For one of our social networks, we saved 4 days.

Chapter 3: Regularizing Centrality Computations

The centrality metrics play an important role in network and graph analysis since they are related with several concepts such as reachability, importance, influence, and power [48, 93, 121, 136, 164]. Betweenness and closeness (BC and CC) are two such metrics. However, the complexity of the best algorithms to compute them are unbearable for today's large-scale networks: for unweighted networks, it is $\mathcal{O}(nm)$ where n is the number of vertices and m is the number of edges in the corresponding graph [29]. For weighted networks, the complexity is more, $\mathcal{O}(nm + n^2 \log n)$. Although this already makes the problem hard even for medium-scale graphs, considering million- and even billion-scale ones, it is clear that we need efficient high performance computing (HPC) techniques.

3.1 Introduction

There are several GPU-based algorithms and parallelization techniques for computing betweenness [90, 148, 164, 135] and closeness [90, 164] centrality. However, as we will show in this work, since these techniques process only a single graph traversal at a time and employ pure fine-grain parallelism, they cannot fully utilize

the GPU and reach the device's peak performance. In addition to these studies, parallel breadth-first search (BFS), which is the main building block to compute closeness centrality values, has been widely studied on shared-memory systems such as GPUs [84, 115, 122] and Intel Xeon Phi [158]. Since these work focus on the parallelization of a single BFS, their natural extension to CC will yield the iterative execution of a fine-grain parallel CC kernel responsible from a single graph traversal. In this work, we propose novel and efficient algorithms and techniques to compute betweenness centrality on GPU and closeness centrality on GPU and Intel Xeon Phi. Although we agree that fine-grain parallelism is still necessary due to the memory restriction of the cutting-edge many-core architectures at hand, we leverage the potential of the hardware by enabling a hybrid coarse/fine-grain parallelism technique that executes multiple simultaneous BFSs.

Although many of the existing techniques leverage parallel processing, one of the most common parallelism available in almost all of today's recent processors, namely instruction parallelism via vectorization, is often overlooked due to nature of the sparse graph kernels. Graph computations are notorious for having irregular memory access pattern, and hence for many kernels that require a single graph traversal, the available vectorization support, which is a great arsenal to increase the performance, is usually considered not very effective. It can still be used, for a small benefit, at the expense of some preprocessing that involves partitioning, ordering and/or use of alternative data structures. To exploit its full potential and enable it for simultaneous

graph traversal approach, we provide an ad-hoc CC formulation based on bitwise operations and propose hardware and software vectorization for that formulation on cutting-edge hardware. Our approach for closeness centrality serves as an example to show how vectorization can be utilized for graph kernels that require multiple BFS traversals. As a result, we experimentally show that compared to the existing solutions, the proposed techniques can be significantly faster while computing exact betweenness and closeness centrality values, on the same device, i.e., without using an additional hardware resource. Furthermore, the proposed techniques can also be used to compute approximate BC and CC values for which the graph traversals are only initiated from a subset of vertices.

The rest of the chapter is organized as follows: Section 3.2 presents a summary of the existing parallelization approaches including accelerator-based algorithms for betweenness and closeness centrality. The proposed parallelization algorithms and techniques are explained in Section 3.3 and their performance is evaluated in Section 3.4. Section 3.5 concludes the chapter.

3.2 Parallelism for network centrality

Background on the closeness and betweenness centrality computation are given in Section 2.2 of Chapter 2. Here, we summarize the existing work on parallel approaches for centrality computation.

The centrality computations can be parallelized in two ways: coarse- and fine-grain. In coarse-grain parallelism, the BFSs are shared among the threads, i.e., a

shortest-path graph is constructed by a single thread. Hence, the threads need to work with separate memory regions in SeqCC and SeqBC, e.g., σ, δ, `pred`, *queue*, *stack*, and `d`. In fine-grain parallelism, a BFS is concurrently executed by multiple threads. Ligra [165] and SNAP [166] are two state-of-the-art shared-memory graph processing frameworks, both make use of fine-grain parallelism for BC computation and will serve as baseline for our BC parallelization techniques. In fine-grain parallelism, although the memory footprint is less compared to the coarse-grain parallelism, concurrency can bring a significant overhead due to the necessity of (relatively) expensive tools such as atomic operations and conflict resolution. That being said, for devices with restricted memory and large potential for concurrent execution, such as GPUs, a fine-grain parallelism is usually necessary.

There are existing studies on computing closeness and betweenness centrality using GPUs; Shi and Zhang developed a software package to do biological network analysis [164]. Later, various parallelism techniques on GPUs for BC and CC computations are experimented by Jia et al. [90]. Concurrently, Pande and Bader studied computing BC of small-world networks on a GPU [135]. Recently, a modified graph storage scheme to obtain better speedups compared to existing solutions is presented [148], which is also a part of this dissertation. Apart from the centrality computation, Merrill et al. [122] propose different fine-grain parallelization techniques for BFS computation, which is a building block for BC and CC computations, and prove to be the fastest solution on GPU architectures. All these studies employ a

pure fine-grain parallelism and *level-synchronized* BFSs. That is, while traversing the graph, the algorithms initiate a GPU kernel for each level ℓ to visit the vertices/edges on that level and find the vertices on level $\ell+1$. One interesting work which combines fine-grain and coarse-grain parallelism is [124]. Although we use the same combination, in [124], a queue-based implementation is employed and hence the number of simultaneous BFSs is limited due to the contention in the queue. Our work alleviates this problem by not employing a queue.

Another recent work on betweenness centrality computation investigates the edge and node parallelism on dynamic betweenness centrality computation [120]. In a preliminary version of this paper, we introduced vectorization support for efficient closeness centrality computation [154]. Other than that, to the best of our knowledge, there is no prior work on computing centrality using hardware and/or software vectorization.

In an earlier work, Saule and Çatalyürek had presented an early evaluation of the scalability of several variants of BFS algorithm on Intel Xeon Phi coprocessor using a pre-production card in which they had presented a re-engineered shared queue data structure for many-core architectures [158]. In another study, they had also investigated the performance of SpMV on Intel Xeon Phi coprocessor architecture, and show that memory latency, not memory bandwidth, creates a bottleneck for SpMV on Intel Xeon Phi [159].

A similar problem to centrality computation is all-pairs shortest path computation. There are several studies on GPU-based parallelization of this problem [94, 117, 131]. A shared memory cache efficient GPU implementation to solve transitive closure and the all-pairs shortest-path problem on directed graphs for large datasets is proposed in [94]. Their solution is able to handle graph sizes that are larger than the DRAM memory of available on the GPU. Matsumoto et al. [117] proposed a blocked algorithm for all-pairs shortest path problem to be used in a hybrid CPU-GPU system where the communication between CPU and GPU is minimized. In [131], the authors present an algorithm to accelerate the all-pairs shortest path computation on GPUs by solving multiple single source shortest path problems at a time, allowing to efficiently access graph data by sharing the data between processing elements in the GPU. In our work, we focus on faster GPU parallelization of betweenness and closeness centrality computation on unweighted graph which have more regularity than the all-pairs shortest path computation allowing finer synchronization and optimization.

3.2.1 Graph storage schemes and parallelization

Many sparse matrix and graph algorithms such as sparse matrix-vector multiplication (SpMV) and BFS are known to be memory bound. Hence, the speedup one can achieve with a GPU significantly depends on to the irregularities in the graph such as the connectivity pattern and degree distribution which can significantly damage load balancing and memory coalescing. Thus, it may be beneficial to store the graph

in the most suitable format that yields a better regularization of computation and memory usage, hence a better performance.

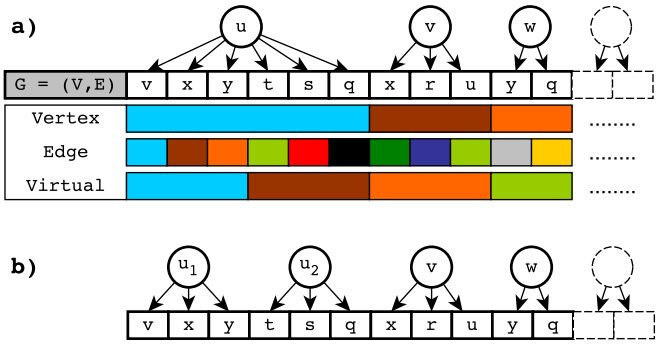


Figure 3.1: a) Vertex-, edge-, and virtual-vertex-based parallelization for centrality computation and the distribution of work to GPU threads which are shown with different colors. $\Delta = 3$ for virtual-vertex-based parallelization. b) The graph structure with virtual vertices.

There are three parallelization techniques that have been used and experimented for closeness and betweenness centrality computations; vertex-based [90, 164], edge-based [90], and virtual-vertex-based [148]. The difference between these techniques is the granularity of the parallelism which impacts both load balancing and the need for synchronization. One of the common storage format for graphs is compressed adjacency list, where the adjacency lists of the vertices are stored consecutively with a secondary pointer array that keeps the start/end pointers which require $(m+n+1)$ values in total. Another commonly used format stores the endpoints of each edge

individually, hence $2m$ values are required. To ease the memory accesses, vertex-based parallelism uses the former and each thread processes an adjacency list at each kernel execution throughout a level-synchronized BFS. On the other hand, the edge-based parallelism uses the latter and each thread processes only a single edge.

As Jia et al. shows, the vertex-based parallelism on GPU suffers from load balancing especially for the graphs with skewed degree distributions [90]. On the other hand, the edge-based parallelism uses more memory and more atomic operations. Sarıyüce et al. proposed the virtual-vertex-based parallelism which simply replaces a problematic, high-degree vertex u with $n_\Delta(u) = \lceil \Gamma(u)/\Delta \rceil$ virtual vertices each having at most Δ edges. That is $n_\Delta(u)$ threads are responsible from processing the edges of a vertex u. Figure 3.1 summarizes how the threads process the edges in vertex-, edge-, and virtual-vertex-based parallelization for centrality computation. In the figure, different threads are shown with different colors and $\Delta = 3$ is used. For betweenness centrality, we also followed the virtual-vertex idea on GPU since as also stated in [148], it performed better than the other techniques in our preliminary BC experiments. We will use n_Δ for the number of virtual vertices and $adj_\Delta(u_*)$ to denote the adjacency list of the arbitrary virtual vertex u_* for an original vertex u in G. For CC experiments, where we use hardware/software vectorization, we will use the compressed adjacency list format and vertex-based parallelism, since with vectorization multiple simultaneous BFSs, the load balancing problem is resolved almost automatically as we will describe later.

Simply speaking, *virtual-vertex-based parallelism* replaces a problematic, high-degree vertex u with $n_\Delta(u) = \lceil \Gamma(u)/\Delta \rceil$ virtual vertices each having at most Δ edges. That is $n_\Delta(u)$ threads are responsible from processing the edges of a vertex u; and these threads have the same amount of work which improves the load balance. Figure 3.1 summarizes how the threads process the edges in vertex-, edge-, and virtual-vertex-based parallelization for centrality computation. In the figure, different threads are shown with different colors and $\Delta = 3$ is used. One can see that the load is imbalanced when *vertex-based* parallelism is used. The load is balanced using *edge-based* parallelism but the granularity of the computation is very fine which increases the synchronization cost (number of atomic operations). Using virtual vertices, each thread has a more balanced amount of work and overall less synchronization (atomic operations) are required. More details are available in [148].

For betweenness centrality, we use virtual-vertex parallelism on GPU since it performed better than the other techniques in our preliminary experiments [148]. We will use n_Δ for the number of virtual vertices and $adj_\Delta(u_*)$ to denote the adjacency list of the arbitrary virtual vertex u_* for an original vertex u in G. For closeness centrality, where we use hardware/software vectorization, we will use the compressed adjacency list format and vertex-based parallelism, since with vectorization multiple simultaneous BFSs, the load balancing problem is resolved almost automatically as we will describe later.

3.3 Faster Network Centrality

Surprisingly, all the existing algorithms proposed for betweenness and closeness centrality prefer a pure fine-grain parallelism that employs an iterative execution of a kernel responsible from a single parallel graph traversal. This approach makes sense for accelerators, since they are memory restricted especially considering the size of today's large scale networks. Hence, a coarse-grain approach in which each thread executes a single BFS is unfeasible, and fine-grain BFSs are almost necessary for the device. Yet, an immediate question still needs to be answered: why only one fine-grain BFS at a time? We believe that there is no valid answer. Furthermore, as we will show, doing otherwise can significantly enhance the BC and CC performance without using any additional hardware resource or one with different characteristics.

3.3.1 A More Regular and Denser Betweenness Centrality Kernel on GPU

For GPU-based BC, we propose a novel parallelization technique which employs simultaneous BFSs where each thread is responsible for processing a (virtual) vertex in a single BFS. On a GPU, there are several ways to do that including manually partitioning the threads for the BFSs or using concurrent streams. In this work, we use interleaved BFSs to achieve a better memory access pattern.

As stated above, all the existing studies that focus on parallel centrality computations employ level-synchronized BFSs: the ℓth kernel execution is responsible from the ℓth level of the BFS, visits the vertices on it, and processes the corresponding

adjacency lists to find the vertices in the $\ell + 1$th level to update their distance information. Note that there are at most L such kernel executions where L is the diameter of the shortest path graph, i.e., the longest distance from the BFS source to a vertex in G. However, the adjacency list of a specific (virtual) vertex u will be processed only in one of these L kernel executions. For the other $L - 1$ executions, the thread responsible for vertex u (u_*) in that BFS may be forced to wait for another thread in the same warp.

Algorithm 7: VirBC ($G = (V, E)$)

```
1  ···
2  ℓ ← 0
   ▷Forward phase
3  visited ←true
4  while visited =true do
5      visited ← false
       ▷Forward-step kernel
6      for each thread t in parallel do
7          if t ≤ n_Δ then
8              u_* ← t                                          ▷virtual vertex
9              u ← the vertex in G corresponding to u_*
10             if dst[u] = ℓ then
11                 for each v ∈ adj_Δ(u_*) do
12                     if dst[v] = ∞ then
13                         dst[v] ← ℓ + 1, visited ← true
14                     if dst[v] = ℓ + 1 then
15                         σ[v]←σ[v] + σ[u]                     ▷atomic
16     ℓ ← ℓ + 1
17 ···
   ▷Backward phase
18 ···
```

Algorithm 7 shows the baseline GPU implementation for the forward phase of Brandes' betweenness centrality algorithm that starts from level $\ell = 1$ and ends at $\ell = L$ when no new vertex is *visited* in the previous kernel execution (the backward phase starts with level $\ell = L$ and stops at $\ell = 1$, and has a similar structure). As described above, this baseline may not utilize the warps efficiently especially for the networks with a large diameter. In fact, the virtual-vertex parallelism solves this problem up to some level when the degrees in the network are considerably larger than Δ. In this case, the consecutive threads will be responsible for the virtual vertices u_* each having Δ edges and coming from the same origin vertex u. Hence, the threads responsible from these virtual vertices will perform essentially the same amount of operation at the same time independently from the BFS source. Since there is less thread divergence, the warps, so the device, will be utilized more effectively.

Using virtual vertices is not a "be all and end all" solution to accelerate BC on a GPU. As we will show in the experiments, the performance it yields is not close the peak performance of the device for many cases. There are several reasons for this low performance: first, when the average degree in the network is low, which may be the case for many sparse networks, its impact on the execution scheme is minimal and considering its overhead, it can be even negative. Furthermore, virtual-vertex-based parallelism does not regularize the uncoalesced memory access pattern which is usually the most important problem of the memory-bounded GPU-based algorithms on sparse matrices, graphs, and networks.

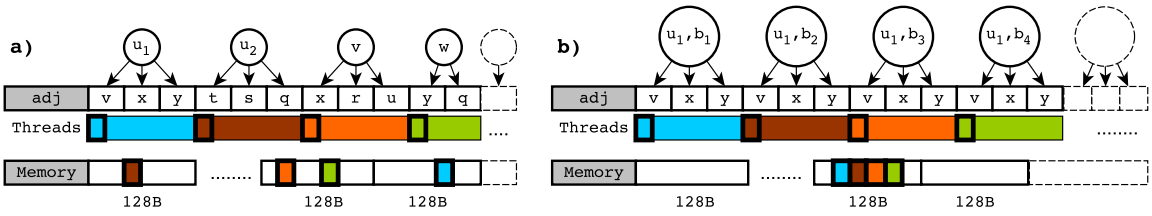


Figure 3.2: A toy example given to show the uncoalesced and coalesced memory access patterns of the virtual-vertex-based scheme (left) and the proposed approach (right) respectively. On the left, three memory transactions are required whereas on the right a single transaction is sufficient (assuming the virtual vertex u_1 is on the same level in all the BFSs).

In a GPU, the threads in half-warps coordinate global memory accesses into a single transaction. If these accesses are uncoalesced (coalesced to many memory blocks), the required information is transferred via multiple 32B, 64B, and 128B transactions which drastically reduce the performance. Consider the toy example in Fig. 3.2(a), where 4 consecutive threads in the same warp (and in the same half-warp) visit their virtual vertices and process the first (neighbor) vertices in the corresponding adjacency lists. Since these adjacency lists are different, the memory locations the threads access, e.g., `dst`[.], can be in different blocks. Considering how level-synchronized BFSs work, 3 transactions are required for the coordinated memory access in Fig. 3.2(a).

The key idea in this work is deviating from the pure fine-grain, single-BFS parallelism to a hybrid, coarse/fine-grain parallelism with a motivation to regularize memory access patterns by employing multiple BFSs in batches. For GPU-based BC,

we aim for sets of consecutive threads that process a (virtual) vertex simultaneously for multiple BFSs. That way the memory access patterns will gain some regularity in BC kernels which, typically, a single parallel BFS lacks.

Algorithm 8: VirBC-Multi ($G = (V, E)$)

```
   ▷B: number of BFSs performed in a batch
1  ···
2  ℓ ← 0
   ▷Forward phase
3  visited ← true
4  while visited = true do
5      visited ← false
       ▷Forward-step kernel
6      for each thread t in parallel do
7          if t ≤ B × n_Δ then
8              u_* ← ⌈t/B⌉                                   ▷virtual vertex
9              b ← t mod B                                   ▷BFS id
10             u ← the vertex in G corresponding to u_*
11             ω_u ← u × B + b
12             if dst[ω_u] = ℓ then
13                 for each v ∈ adj_Δ(u_*) do
14                     ω_v ← v × B + b
15                     if dst[ω_v] = ∞ then
16                         dst[ω_v] ← ℓ + 1, visited ← true
17                     if dst[ω_v] = ℓ + 1 then
18                         σ[ω_u] ← σ[ω_v] + σ[ω_u]          ▷atomic
19     ℓ ← ℓ + 1
20 ···
   ▷Backward phase
21 ···
```

Let $\mathcal{B}$ be the number of BFSs in a batch. One can execute a kernel with $\mathcal{B} \times n_\Delta$ threads where the ith BFS is executed by the n_Δ threads starting from the thread $(i-$

1) $\times n_\Delta$. However, this only handles the work in less kernel calls without regularizing the memory accesses. For this reason, we employ interleaved BFSs. Algorithm 8 implements the idea for the forward phase of BC. Its most important difference from Algorithm 7 lies within the memory accesses to the arrays `dst`[.] and σ[.]: in VirBC, the neighbor vertex ids have a very high impact in the locality of memory accesses by consecutive threads. Since they can be different, the blocks that need to be accessed by a half-warp can be at various places of the memory. On the other hand, in VirBC-Multi, the memory index ω_v computed for a vertex/BFS pair will differ only by one for two consecutive threads processing the same virtual vertex. Hence, $\mathcal{B}$ consecutive threads will access consecutive memory locations and a single transaction may be sufficient for the coordinated memory access as Fig 3.2(b) shows for our toy example. In Algorithm 8, the arrays `dst`[.] and σ[.] are of length $\mathcal{B} \times n$. Hence, except the graph G, the memory footprint of VirBC-Multi is $\mathcal{B}$ times larger than that of VirBC which is one of the drawbacks of our solution. That being said, a small $\mathcal{B}$ would be sufficient to increase the performance as the experiments will show.

We are aware that a vertex will not appear exactly in the same level for all $\mathcal{B}$ BFSs in a batch. Hence, although $\mathcal{B}$ consecutive threads are responsible from the same (virtual) vertex, it is not guaranteed that all these threads will process the adjacency lists in the same kernel execution. But even in the original virtual-vertex-based scheme with a single BFS execution, such a guarantee was there only for the consecutive virtual vertices generated from the same original vertex. For the warps

processing such vertices, our modification on the parallelism can be considered as a trade-off of warp utilization (occupancy) and the ratio of coalesced memory accesses. On the other hand, for the warps which already process the original vertices with degree Δ or less, the utilization will probably not be harmed. Furthermore, recent studies show that most of the vertices in G appear only in the middle levels of any BFS for the networks with small-world properties, which typically includes social networks [25, 153]. Thus the proposed scheme can even increase the warp occupancy.

The overhead of the proposed technique increases when the maximum level L for the BFSs fluctuates, i.e., when the variance of their distribution is high. If this is the case the BFSs in a pack that are already completed will stay in the process and wait for the one in the pack with the highest L value. Fortunately, the real-life networks exhibit small-world network characteristics and have small diameters, hence L does not fluctuate for the BFSs.

3.3.2 A More Regular and Denser Closeness Centrality Kernel on GPU and Intel Xeon Phi

Having irregular memory access and computation that prevent a proper vectorization is a common problem of sparse kernels. The most emblematic sparse computation is certainly the multiplication of a sparse matrix by a dense vector (SpMV). In SpMV, the problem of improving vector-register (also called *SIMD register*) utilization and regularizing the memory access pattern was deeply studied and methods such as register blocking [36, 175] or by using different matrix storage formats [26, 111] have

been proposed. Arguably, the most efficient method to regularize the memory access pattern is to multiply a sparse matrix by multiple vectors if this is possible. When the multiple vectors are organized as a dense matrix, the problem becomes the multiplication of a sparse matrix by a dense matrix (SpMM). While each nonzero of the sparse matrix causes the multiplication of a single element of the vector in SpMV, it causes the multiplications of as many consecutive elements of the dense matrix as its number of columns in SpMM.

Adapting that idea in closeness centrality essentially boils down to the computing multiple sources at the same time, simultaneously. But contrarily to SpMV, where the vector is dense hence each non-zero induces exactly one multiplication, in BFS, not all the non-zeros will induce operations. In other words, a vertex in BFS may or may not be traversed depending on which level is currently being processed. Therefore, the traditional queue-based implementation of BFS does not seem to be easily extendable to support multiple BFSs in a vector-friendly manner.

An SpMV-based formulation of closeness centrality

The main idea is to revert to a more basic definition of level synchronous BFS traversal. Vertex v is part of level ℓ if and only if one of the neighbor of v is part of level $\ell - 1$ and v is not part of any level $\ell' < \ell$. This formulation is commonly used in parallel implementation of BFS on GPU [90, 135, 164] but also in some shared memory [3] and distributed memory implementations [35].

The algorithm is better represented using binary variables. Let x_i^ℓ be the binary variable that is `true` if vertex i is part of the frontier at level ℓ for a BFS. The neighbors of level ℓ is represented by a vector $y^{\ell+1}$ computed by

$$y_k^{\ell+1} = \texttt{OR}_{j \in \Gamma(k)} x_j^\ell.$$

The next level is then computed with

$$x_i^{\ell+1} = y_i^{\ell+1} \texttt{ AND not } (\texttt{OR}_{\ell' \leq \ell} x_i^{\ell'}).$$

Using these variables, one can update the closeness centrality value of vertex i by adding $\frac{x_i^\ell}{\ell}$ if i is at level ℓ. One can remark that $y^{\ell+1}$ is the result of the "multiplication" of the adjacency matrix of the graph by x^ℓ in the (`OR`,`AND`) semi-ring.

Implementing BFS using such an SpMV-based algorithm changes its asymptotic complexity. The traditional queue-based BFS algorithm has a complexity of $\mathcal{O}(|E|)$. But the complexity of the SpMV-based algorithm described above depends on how the adjacency matrix is stored. If it is stored column-wise, then it is easy to traverse column j only if the value of x_j^ℓ is `true`. This leads to an $\mathcal{O}(|E|)$ implementation of BFS, and such an implementation is not essentially different from the queue-based implementation of BFS: they both follow a top-down approach. However, when x_j^ℓ is `true`, the updates on the entries of $y^{\ell+1}$ vector cause scattered writes to memory which are problematic when executed in parallel.

On the other hand, by storing the adjacency matrix row-wise, different values of x^ℓ are gathered to compute a single element of $y^{\ell+1}$. This yields a bottom-up

implementation of BFS which has a natural write access pattern. However, it becomes impossible to only traverse the relevant nonzero of the matrix and the complexity of the algorithm becomes $\mathcal{O}(|E| \times L)$, where L is the diameter of the graph. This is the implementation that we favor and we do not feel that this asymptotically worse complexity is a problem since it has been noted many times before that social networks have small world properties. So, their diameter tends to be low. Note that the small world property only informs on the average distance between two vertices is proportional to $\log(|V|)$ while we are interested in the maximum distance. There could be small world graphs with a long chain on where our technique might not apply as gracefully.

An SpMM-based formulation of closeness centrality

It is easy to derive an algorithm from the formulation given above for closeness centrality that processes multiple sources at once (see Algorithm 9). The algorithm processes sources by batches of $\mathcal{B}$. For each level ℓ, it builds a binary matrix x^ℓ where $x^\ell_{i,s}$ indicates if vertex i is at distance ℓ of source vertex s where $0 \leq s < \mathcal{B}$ is the relative source id in the batch. The first part of the algorithm is `Init` which computes x^0.

After `Init`, the algorithm performs a loop that iterates over the levels of the BFSs. The second part is `SpMM` which builds the matrix $y^{\ell+1}$ by multiplying the adjacency matrix with x^ℓ. After each `SpMM`, the algorithm enters its `Update` phase where $x^{\ell+1}$ is

Algorithm 9: CC-SpMM: SpMM-based centrality computation

Data: $G = (V, E)$, $\mathcal{B}$
Output: $\texttt{cc}[.]$
▷Init
$\texttt{cc}[v] \leftarrow 0, \forall v \in V$
$\ell \leftarrow 0$
partition V into k batches $\Pi = \{V_1, V_2, \ldots, V_k\}$ of size $\mathcal{B}$
for each batch *of vertices* $V_p \in \Pi$ **do**
    $x^0_{s,s} \leftarrow 1$ if $s \in V_p$, 0 otherwise
    while $\sum_i \sum_s x^\ell_{i,s} > 0$ **do**
        ▷SpMM
        $y^{\ell+1}_{i,s} = \texttt{OR}_{j \in \Gamma(i)} x^\ell_{j,s}, \forall s \in V_p, \forall i \in V$
        ▷Update
        $x^{\ell+1}_{i,s} = y^{\ell+1}_{i,s}$ $\texttt{AND not}(\texttt{OR}_{\ell' \leq \ell} x^{\ell'}_{i,s}), \forall s \in V_p, \forall i \in V$
        $\ell \leftarrow \ell + 1$
        for all $v \in V$ **do**
            $\texttt{cc}[v] \leftarrow \texttt{cc}[v] + \frac{\sum_s x^\ell_{v,s}}{\ell}$
return $\texttt{cc}[.]$

computed and then the closeness centrality values are updated using the information of level $\ell + 1$.

By letting $\mathcal{B}$ be the size of the vector register of the machine used, a row of the x and y matrices exactly fits in a vector-register, and all the operations become vector-wide `OR`, `AND` and `not` and bit-count operations. Figure 3.3 presents an implementation of this algorithm using AVX instructions ($\mathcal{B} = 256$). We use similar codes to leverage 32-bit integer types, SSE registers and Xeon Phi's 512-bit registers in the experiments. The code uses three arrays to store the internal state of the algorithm. `current` stores

x^ℓ for the current level ℓ, `neighbor` stores $y^{\ell+1}$ and `visited` stores $\texttt{OR}_{\ell' \leq \ell} x^{\ell'}$. The function `bitCount_256`(.) calls the appropriate bit-counting instructions.

A potential drawback of the SpMM variant of the closeness centrality algorithm is that each traversal of the graph now accesses a wider memory range than the one used in an SpMV approach. This can harm the cache locality of the algorithm. To see the impact on cache-hit ratio, we wrote a simulator to emulate the cache behavior during the SpMM operation. The simulator assumes that the computation is sequential; the cache is fully associative; it uses cache-lines of 64 bytes; only the x vector (`current` array in the code) is stored in the cache; and the cache is completely flushed between iterations.

Figure 3.4 presents the cache-hit ratios with a cache size of 512K (the size of Intel Xeon Phi's L2 cache) for different number of BFSs and for the seven graphs we will later use in the experimental evaluation. The cache hit-ratio degrades by about 20% to 30% when the number of concurrent BFSs goes from 32 to 512. This certainly introduces a significant overhead, but we believe it should be widely compensated by reducing the number of iterations of the outer loop by a factor of 16.

Software vectorization

The hardware vectorization of the SpMM kernel presented above limits the number of concurrent BFS sources to the size of the vector registers available on the architecture. However, there is no reason to limit the method to the size of a single register. One could use two registers instead of one and perform twice more sources

```
void cc_cpu_256_spmm (int* xadj, int* adj, int n, float* cc)
{
  int b = 256;
  size_t size_alloc = n * b / 8;
  char* neighbor = (char*)_mm_malloc(size_alloc, 32);
  char* current = (char*)_mm_malloc(size_alloc, 32);
  char* visited = (char*)_mm_malloc(size_alloc, 32);
  for (int s = 0; s < n; s += b) {
    //Init
    ...
    int cont = 1, level = 0;
    while (cont != 0) {
      cont = 0; level++;
      //SpMM
#pragma omp parallel for schedule (dynamic, CC_CHUNK)
      for (int i = 0; i < n; ++i) {
        __m256 vali = _mm256_setzero_ps();
        for (int j = xadj[i]; j<xadj[i+1]; ++j) {
          int v = adj[j];
          __m256 state_v = _mm256_load_ps((float*)(current + 32 * v));
          vali = _mm256_or_ps (vali, state_v);
        }
        _mm256_store_ps ((float*)(neighbor + 32 * i), vali);
      }
      //Update
      float flevel = 1.0f / (float) level;
#pragma omp parallel for schedule (dynamic, CC_CHUNK)
      for (int i = 0; i < n; ++i) {
        __m256 nei = _mm256_load_ps ((float *)(neighbor + 32 * i));
        __m256 vis = _mm256_load_ps ((float *)(visited + 32 * i));
        __m256 cu = _mm256_andnot_ps (vis, nei);
        vis = _mm256_or_ps (nei, vis);
        int bcnt = bitCount_256(cu);
        if (bcnt > 0) {
          cc[i] += bcnt * flevel; cont = 1;
        }
        _mm256_store_ps ((float *)(visited + 32 * i), vis);
        _mm256_store_ps ((float *)(current + 32 * i), cu);
      }
    }
  }
  _mm_free(neighbor); _mm_free(current); _mm_free(visited);
}
```

Figure 3.3: Hardware vectorization using AVX for the SpMM-based formulation of closeness centrality.

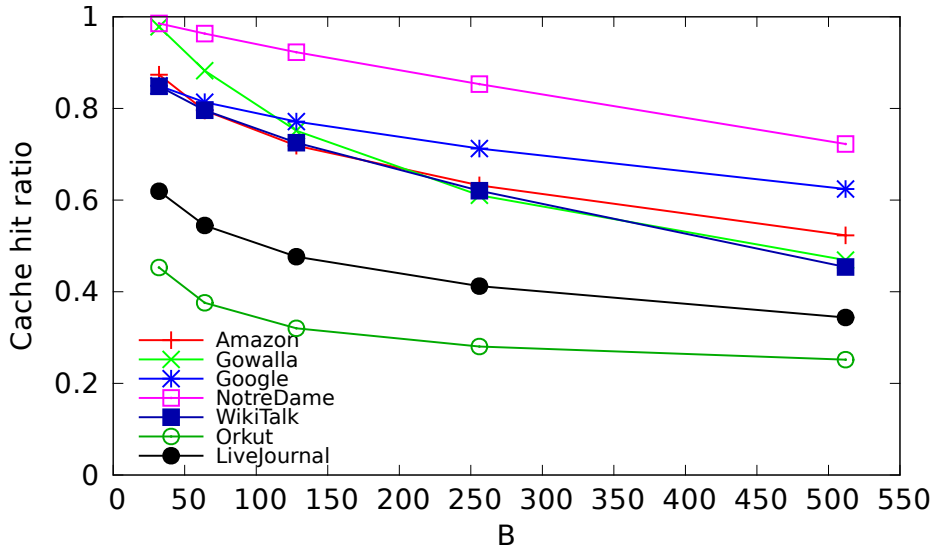


Figure 3.4: Simulated cache-hit ratio of the SpMM variant on a 512K cache (e.g., Intel Xeon Phi's L2 cache).

at once. The penalty on the cache locality will certainly increase, but probably not by a factor of two.

Since we want to try various number of simultaneous BFS, the implementation effort for manual vectorization of each version becomes prohibitive. Therefore, we developed a unique code that allows to easily change the number of concurrent source traversed. Figure 3.5 presents a fragment of this code which has been carefully written to allow the compiler to leverage vector instructions where possible. The key of this code is to specify the number of simultaneous traversals as a C++ template parameter instead of using a function parameter. This forces the compiler to generate a different

object code for each value of the template parameter `vector_size` (expressed in number of 32-bit words). Therefore, it allows the compiler on a CPU architecture to utilize the SSE instructions if `vector_size` is 4 or to utilize the AVX instructions if it is more than 8. The right template parameter is selected in a wrapper function (not shown here).

Instead of using explicit registers, this compiler vectorized code expresses the state of the x vector as an array of 32-bit integers. The compiler is hinted at unrolling these accesses to prevent a loop and expose their vectorial nature. Though, the C++ language does not directly allow that vectorization to take place because the various pointers of the function might point to overlapped memory. The `__restrict__` language extension is used to instruct the compiler that none of the arrays will ever overlap, allowing the compiler to generate the vector instructions when it believes that they are appropriate. As the experiments will show, the compiler-based vectorization in Figure 3.5 perform almost as good as the manually vectorized code given in Figure 3.3 which is useful in practice since the compiler-based vectorization is much more flexible to change the number of simultaneous BFSs.

Closeness centrality on GPU

The SpMM-based approach for closeness centrality can be directly adapted for GPU since the hardware is already modeled for SIMD execution. Simply put, one can consider one operation on a CUDA warp as one Xeon Phi SIMD operation. For our implementation, we used 64-bit integers to store parts of `current`, `neighbor`, and

```
template<int vector_size>
void cc_cpu_spmm_soft_vec_t (int* __restrict__ xadj,
                             int* __restrict__ adj,
                             int n, float* __restrict__ cc)
{
  int b = vector_size * 32, n_align = b / 8;
  size_t size_alloc = n * b / 8;
  char* __restrict__ neighbor = (char*)_mm_malloc(size_alloc, n_align);
  char* __restrict__ current = (char*)_mm_malloc(size_alloc, n_align);
  char* __restrict__ visited = (char*)_mm_malloc(size_alloc, n_align);
  for (int s = 0; s < n; s += b) {
    //Init
    ...
    int cont = 1, level = 0;
    while (cont != 0) {
      cont = 0; ++level;
      //SpMM
#pragma omp parallel for schedule (dynamic,CC_CHUNK)
      for (int i = 0; i < n; ++i) {
        int vali[vector_size];
#pragma unroll
        for (int k = 0; k < vector_size; ++k)
          vali[k] = 0;
        for (int j = xadj[i]; j < xadj[i+1]; ++j) {
          int v = adj[j];
#pragma unroll
          for (int k = 0; k < vector_size; k++)
            vali[k] = vali[k] | ((int*)current)[v*vector_size+k];
        }
#pragma unroll
        for (int k = 0; k < vector_size; ++k)
          ((int*)neighbor)[i*vector_size+k] = vali[k];
      }
      //Update
      ...
    }
  }
  _mm_free(neighbor); _mm_free(current); _mm_free(visited);
}
```

Figure 3.5: Compiler vectorization for the SpMM-based formulation of closeness centrality.

`visited` arrays per thread. Thus, each thread can use a bitwise operation to process 64 BFSs simultaneously. When a vertex is assigned to a single GPU warp (containing 32 threads), $\mathcal{B} = 32 \times 64 = 2,048$ BFSs can be handled simultaneously. For memory-bound kernels such as a graph traversal, only a half-warp (16 threads) may also be considered as a counterpart of a SIMD operation on Xeon Phi, since the GPU coordinates the global memory accesses of the threads in a half-warp into a single transaction. Or similar to software vectorization, one can go wider and use more than a warp per vertex to support more than $2,048$ BFSs. We experimented with these three options and assign a vertex to 16, 32, and 64 threads. A similar direction one can follow to increase $\mathcal{B}$ is assigning more work to each thread. That is, by doubling the work of a single thread and assigning 128 BFSs to a thread and a vertex to a warp, one can handle $\mathcal{B} = 4,096$ BFSs at once, and hence, halves the number of kernel executions. However, while following any of these approaches, we are always limited by the memory footprint of the kernel which is a problem for a memory-restricted device such as GPU.

For our GPU-based CC implementation, we used the traditional compressed adjacency list format instead of virtual-vertices we employed in our GPU-based BC implementation. As explained in Section 3.2.1, virtual vertices are proposed to improve the load balance inside a CUDA warp for a single BFS. Since each thread is responsible for a single BFS and when a vertex is not on the current level ℓ of the corresponding BFS, the thread needs to wait the others in the warp. However, in

our CC implementation, since a thread is responsible for multiple BFSs (i.e., 64 of them) it is more likely that at least in one of these BFSs, the thread will need to work. Thus, warp occupancy is expected to be high for the SpMM-based CC. Furthermore, when a single vertex is assigned to a warp, each thread will visit the same adjacency list. Thus, there will not be a load balancing problem and the memory accesses will be highly coalesced. In our experiments, we compared the performance of the SpMM-based implementation (`GPU-SpMM`) with the virtual-vertex-based ones with one BFS at a time (`GPU-VirCC`) and multiple BFSs (`GPU-VirCC-Multi`), where the latter adapts the parallelization techniques we explained for BC in Section 3.3.1.

Implementation details

We improved the performance of the SpMM-based implementation given in Figure 3.3 (as well as the compiler-vectorized one in Figure 3.5 and GPU-based implementation), by employing two simple modifications. In the first modification, which is in the `SpMM` part of Figure 3.3, before traversing the adjacency list of the ith vertex, the algorithm checks that if all the $\mathcal{B} = 256$ `visited` bits corresponding to $\mathcal{B}$ BFSs assigned to the thread were already set to 1 by the previous or current level expansions. If this is the case, since the vertex has already been visited in all the BFSs, the thread skips the `SpMM` part and directly goes to the `Update` part. For the GPU implementation, each thread checks the corresponding 64 bits in the `visited` array. Note that, a warp in the SpMM kernel can terminate only when the $32 \times 64 = 2,048$ `visited` bits are already equal to 1.

The second modification is similar to the first one but this time it is in the *Update* part of Figure 3.3: when all the $\mathcal{B}$ bits in the `visited` array were already set to 1, the code sets the corresponding `current` bits to 0 and ends the `Update` part without any other bitwise operations or bit counting. Similar to the first modification, in the GPU-based CC implementation, each thread in a warp takes this shortcut by using the 64 bits corresponding to the `visited` information of the 64 BFSs and sets the corresponding 64 bits in the `current` array to 0.

3.4 Experiments

The experiments were carried out on a system equipped with two Intel Sandybridge-EP CPUs clocked at 2.00Ghz and 256GB of memory split across two NUMA domains. Each CPU has eight-cores (16 cores in total) and HyperThreading is enabled. Each core has its own 32kB L1 cache and 256kB L2 cache. The 8 cores on a CPU share a 20MB L3 cache. The machine is equipped with an NVIDIA Tesla K20c GPU featuring 13 Streaming Multiprocessors, 192 cores per SM clocked at 700 MHz (for a total of 2496 CUDA cores), and 4.8GB of global memory clocked at 2.6 GHz. ECC is enabled. The system also has an Intel Xeon Phi coprocessor with 8 memory controllers and 61 cores clocked at 1.05GHz. There is a 32kB L1 data cache, a 32kB L1 instruction cache, and a 512kB L2 cache associated with each core. The bandwidth of each core is 8.4GB/s where the cores' memory interface are 32-bit wide with two

channels. Although the cores are expected to provide 512.4GB/s, the bandwidth between the memory controllers and they are limited by the ring network in between which theoretically supports at most 220GB/s.

On the software side, we run a 64-bit Debian with Linux 2.6.39-bpo.2-amd64. All the codes are compiled with GCC with the -O3 optimization flag in version 4.4.4. Xeon Phi codes are compiled with the Intel C++ Compiler in version 13.1 using -O3 optimization flag. CUDA 5.0 is used with flag -arch sm_20. We have carefully implemented all the algorithms using C++. To have a base-line comparison, we implemented OpenMP versions of the CPU-based betweenness and closeness centrality algorithms. Note that, our system has 16 cores. When implementing the CPU based closeness centrality code, we made use of the direction optimization technique, presented in [25]. Other than direction optimization, no particular optimizations have been applied to the CPU codes except the ones performed by the compiler. We also used various studies from the literature to evaluate the practical performance of our GPU-based betweenness centrality implementation with virtual vertices and multiple BFSs and SpMM-based closeness centrality implementation.

For the experiments, we used a set of graphs from the SNAP dataset[1]. Directed graphs were made undirected and the largest connected component is extracted and used in the experiments. The list of graphs and the properties of the largest components that are used in our experiments can be found in Table 3.1.

[1] http://snap.stanford.edu/data/index.html

Graph	$\|V\|$	$\|E\|$	Avg. $\|\Gamma(v)\|$	Max. $\|\Gamma(v)\|$	Diam.
`Amazon`	403K	2,443K	6.0	2,752	19
`Gowalla`	196K	950K	4.8	14,730	12
`Google`	855K	4,291K	5.0	6,332	18
`NotreDame`	325K	1,090K	3.3	10,721	27
`WikiTalk`	2,388K	4,656K	1.9	100,029	10
`Orkut`	3,072K	117,185K	38.1	33,313	9
`LiveJournal`	4,843K	42,845K	8.8	20,333	15

Table 3.1: Properties of the largest connected components of the graph used in the experiments.

All the results presented in this section are computed by using the total application time from the moment where the graph is fully loaded into the main memory of the machine to the moment where the final centrality values are available in the main memory of the node. In particular, the time excludes reading the graph from the hard drive; but it includes the transformations such as virtualization and all the communications between the host and the device. Using these times, we computed the traversed edges per second (TEPS) values and report them on the figures in this section. Given the total application *time* (in seconds) for K sources/BFSs on a graph with m (undirected) edges, the TEPS value is equal to $(m \times K)/time$. Note that to process K sources, the algorithm needs $K/\mathcal{B}$ kernel executions where each of the kernels handles $\mathcal{B}$ sources.

3.4.1 Evaluating the proposed betweenness centrality algorithm VirBC-Multi

In this section, we investigate the efficiency of our virtual-vertex based BC algorithm. We first analyze the VirBC-Multi algorithm with different parameters, then present the absolute numbers on its performance by a comparison with existing work in the literature. We used $\Delta = 8$ for virtualization. Since the computations can be extremely long (months), we did not used all the n BFS sources in the graphs and measured the time for $1,024$ sources/BFSs in total. We observed that the runtimes of single kernel executions to be very stable, allowing us to make a meaningful extrapolation. Thus, if necessary, the results can be used to linearly extrapolate the runtime for the whole graph.

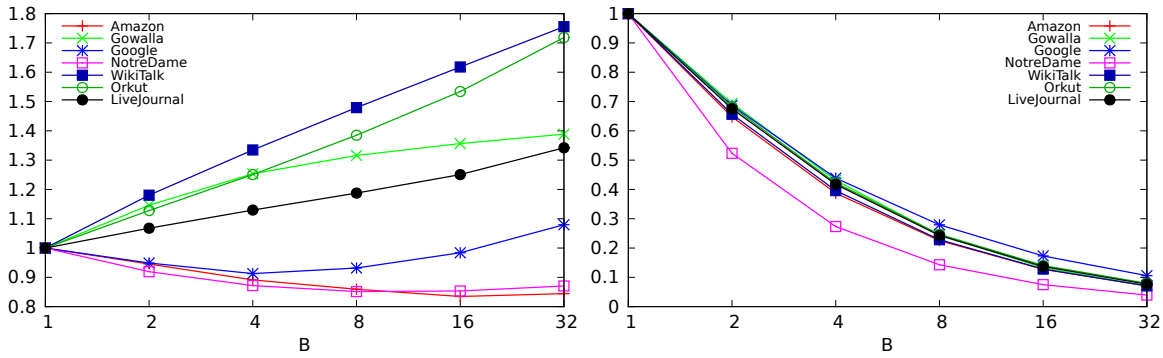
Analysis of VirBC-Multi

We first investigate the validity of one of the assumptions we make: batching multiple traversals is useful because a vertex only appears in a small number of levels. This assumption is expected to lead to a high number of threads within a warp concurrently expending the same vertex. It should improve the computation by increasing the reutilization of the graph data structure and by structuring the memory accesses made by a warp into regular patterns. To verify this, we computed two indices.

The first indice is the number of working warps; in each kernel call, a thread is said to be working if it passes the condition line 12 of VirBC-Multi (Algorithm 8);

that is to say, if that vertex is expended. Similarly, a warp is said to be working if one of its 32 threads passes that line. When $\mathcal{B}$ grows, the structure of the warps change leading to differences in the number of working warps. The more working warps there are, the more computation the GPU will need to perform (this simplification may not be true for all the kernels that run on GPUs, but since we employ virtual vertices for BC, each warp takes essentially the same number of operations which makes the simplification valid). In Figure 3.6(a), we present how the number of working warps within the BC computation is impacted by $\mathcal{B}$. Surprisingly, the number of working warps evolves differently for different graphs. For some graphs, e.g., `Amazon`, `NotreDame`, and `Google`, the number initially decreases but then either stabilizes or increases. For some other graphs, e.g., `WikiTalk`, `Orkut`, `Gowalla`, and `LiveJournal`, the number increases. However, overall, the variation of the number of working warps is fairly small; the decrease is never more than 20% and the increase is never more than 75%. This indicates that batching sources has little impact on thread divergence, but that impact is mostly negative.

The second index measures how many times a virtual vertex is non-simultaneously traversed. When $\mathcal{B} = 1$, each virtual vertex is traversed exactly once per source and each of these traversals is performed by a different warp. But when $\mathcal{B}$ increases then a virtual vertex might be traversed multiple times by a single warp, and we say that these two virtual vertices are traversed simultaneously. What we are interested in, overall, is how many different warps traverse a given virtual vertex. Figure 3.6(b)



(a) Working warps (b) Non simultaneous virtual vertex traversal

Figure 3.6: Analyzing the behavior of VirBC-Multi. The values are normalized relatively to the case $\mathcal{B} = 1$ and accumulated over the iterations of a batch.

shows that the number of non-simultaneous traversals sharply decreases for all the graphs when $\mathcal{B}$ increases. This number improves by more than 85% for all the graphs. This should significantly improve the coalescing of the memory operations performed by various kernels.

We can conclude that the previous increase in the number of working warps is most likely linked to the fact that the warps were naturally well structured because the consecutive vertices are close in the graph and are typically traversed in the same level, thanks to the virtualization. We show the actual impact of varying the number of simultaneous sources $\mathcal{B}$ in performance is in Figure 3.7. All the values are normalized to the time taken by the variant that executes one BFS at a time. A first observation is that all the graphs benefit from executing multiple sources in a batch. But the rate of improvement is different. For instance the improvement

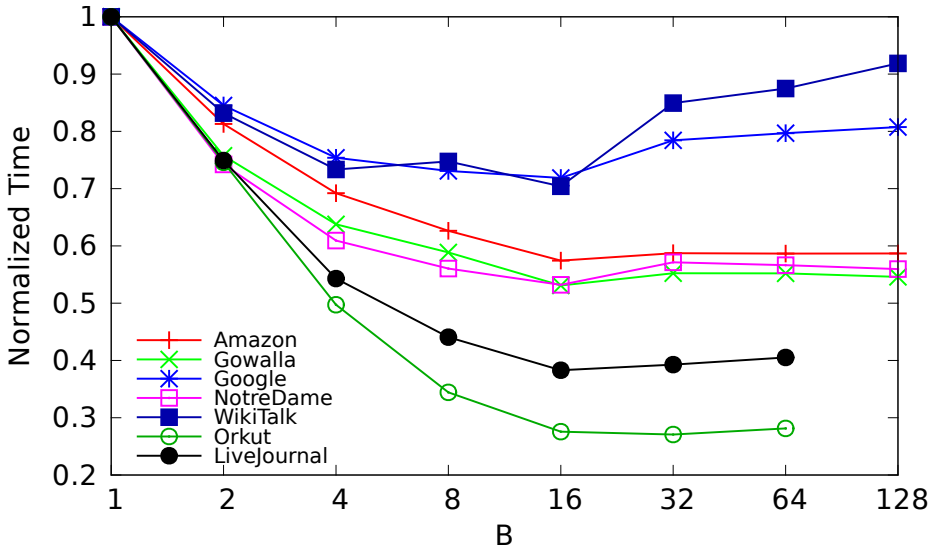


Figure 3.7: Impact of $\mathcal{B}$ on VirBC-Multi run on an NVIDIA Tesla K20

seen on `Orkut` is very similar to the improvement in non-simultaneous virtual vertex traversal (Fig. 3.6(b)). On the other hand, other graphs, such as `Amazon`, incur lesser improvements. Finally for some graphs, the normalized time is V-shaped. We guess that the increase in memory occupation with large $\mathcal{B}$ values is detrimental for some cases. Alternatively, it is possible that for some cases, the memory accesses were already fairly well organized and the proposed techniques only have a limited impact.

Evaluating the absolute performance

We experimentally evaluated the algorithms given in Section 3.3.1 on betweenness centrality kernels. There are mainly three variants: OpenMP-based parallel CPU implementation (Cpu-BC), GPU implementation with virtual vertices (VirBC) and

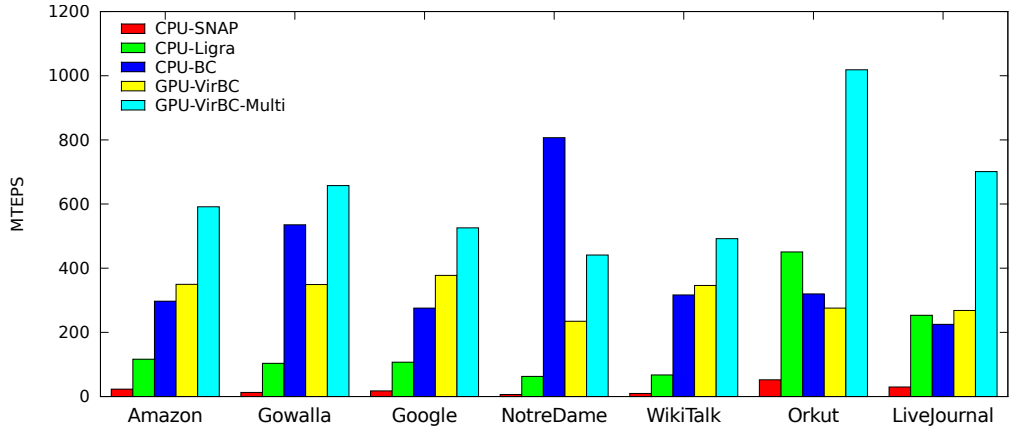


Figure 3.8: Evaluation of the algorithms in terms of MTEPS. The values for the proposed algorithms are the best ones we obtained with different $\mathcal{B}$ values.

VirBC-Multi. We also compared our techniques with the betweenness centrality kernels in the state-of-the-art shared-memory graph processing frameworks Ligra [165] and SNAP [166]. Comparisons are done in terms of *million traversed edges per second* (MTEPS) which is computed as $(1,024 \times |E|)/(10^6 \times time)$, where $|E|$ is the number of (undirected) edges and *time* is the time required to complete all the $1,024$ BFSs we perform for a configuration. For VirBC-Multi, we only report the performance achieved using the best value for $\mathcal{B}$. This value is representative of the performance one can get on a real application since it can easily be discovered at runtime during the first few iterations of the overall algorithm.

Figure 3.8 presents the MTEPS for BC algorithms when executed on the seven networks given in Table 3.1. As Fig. 3.8 shows, VirBC-Multi is superior to the

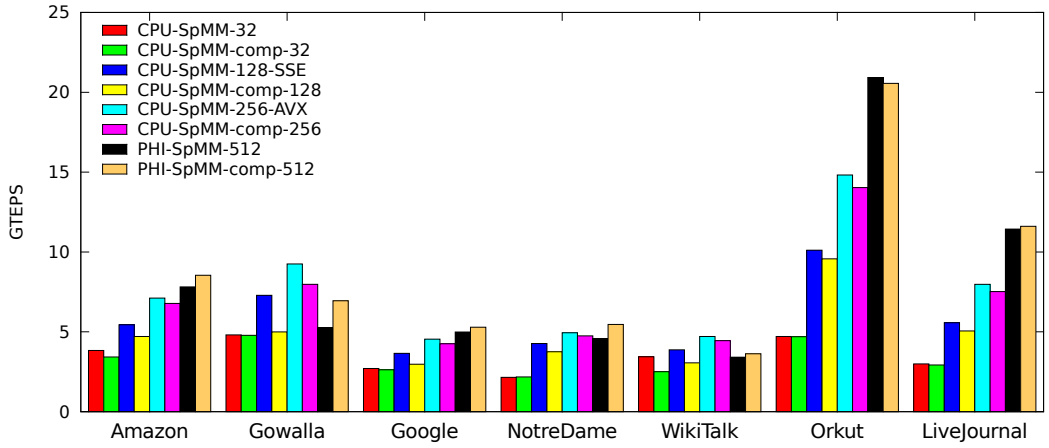


Figure 3.9: The compiler- and manually-vectorized implementation reach similar performance.

others on 6 of 7 graphs. On average, VirBC-Multi is 35 times faster than SNAP, 4.7 times faster than Ligra, 68% faster than Cpu-BC and 96% faster than VirBC. In terms of the performance, VirBC-Multi reaches to 1 GTEPS on `Orkut` network.

3.4.2 Evaluating the proposed SpMM-based closeness centrality algorithm

The closeness centrality experiments are performed using a total of $16,384$ sources. Hence, for a configuration with $\mathcal{B}$ simultaneous BFSs uses $16,384/\mathcal{B}$ kernel executions. Similar to BC experiments, we did not observe a significant variance among the execution times of these executions. The presented TEPS results in the figures are computed by linear extrapolation for the entire graph.

SpMM-based closeness centrality on x86-based architectures

We will first have a look at the performance of our techniques on CPU and Intel Xeon Phi. Since they present similar patterns and Intel Xeon Phi obtains a better performance, we will only present the results for that architecture in this subsection. As a first experiment, we compare the manual and compiler-based hardware vectorization options. For manual vectorization, we implemented 32-bit, 128-bit SSE, 256-bit AVX (see Figure 3.3), and 512-bit Intel Xeon Phi versions with various intrinsics supported by the hardware for a concurrent execution of 32, 128, 256 and 512 BFSs, respectively. For compiler-based vectorization, we used the code (partially) given in Figure 3.5 without the modifications and let the compiler optimize it with -O3 flag. Figure 3.9 gives the performance results in terms of billions of traversed edges per second (GTEPS). The bars in the figure with `-comp` keyword are the ones with the compiler-vectorized versions. For almost all the graphs, 512-bit Intel Xeon Phi vectorization gives the best results. For `Gowalla`, `NotreDame`, and `WikiTalk`, 256-bit versions are better. Overall, manual vectorization is only slightly better than compiler-based vectorization. This shows that if the code is properly written, the compiler does its job and optimizes relatively well. We will mainly use the compiler-vectorized implementation in the rest of the text, since it is more flexible and its performance is comparable to the manually-vectorized one.

The implementation on Intel Xeon Phi uses the *offload mode*. The memory transfer time is optimized by using large memory pages whose size is set with an environment variable `MIC_USE_2MB_BUFFERS` = `4K`. The memory allocation is performed in two phases; as usual in Linux systems, the memory allocation routine is called to allocate the virtual pages and the physical pages are allocated when the memory is touched for the first time. On Intel Xeon Phi, the physical page allocation is fairly slow. To give a point of reference, in our preliminary experiments, the physical memory allocation required to perform $8,192$ BFSs on `Google` (2.44 GB) takes 0.88 seconds; while performing the BFSs takes 1.44 seconds. Still, this overhead increases only with $8,192$ and it does not change with the number of sources used for centrality computation. Hence, considering the number of vertices, n, is much larger than $\mathcal{B}$, the overhead are small compared to the time required to process the whole graph for exact centrality computation (we remark that the presented results are extrapolated taking the memory allocation time into consideration). However, their impact can be higher while using sampling and approximation techniques for closeness centrality.

We experimented with values of $\mathcal{B}$ from 32 to $8,192$ (higher values of $\mathcal{B}$ would lead to out-of-memory for the larger graphs). We present the performance of the compiler-vectorized implementation that benefits from the modifications presented at the end of Section 3.4 in Figure 3.10. In short, the performance increases with $\mathcal{B}$. We can observe that the rate of improvement is higher when hardware vectorization is leveraged, i.e., when $\mathcal{B}$ is smaller than the register size of Intel Xeon Phi, compared to

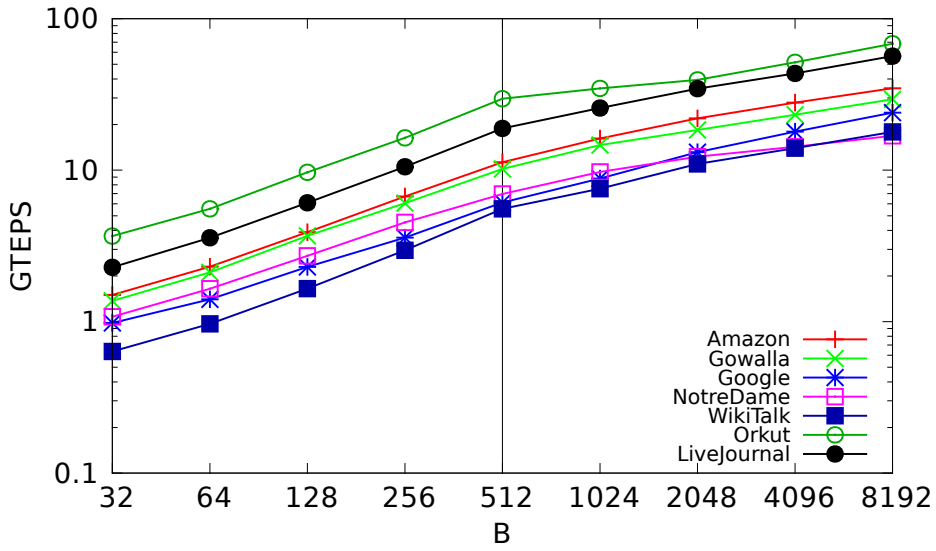


Figure 3.10: Impact of the number of simultaneous BFS on the performance obtained on Intel Xeon Phi with the mofications described in Section 12. The separation between hardware and software vectorization is marked.

the case when when software vectorization is leveraged. Yet, software vectorization still provides significant performance improvements for all the graphs. In the rest of the experiments, we will use the configuration with $8,192$ simultaneous traversals since it obtains best performance.

To put the results into perspective, in Figure 3.11, we compare the performance of the fine-grain BFS technique developed for Intel Xeon Phi presented in [158] (`PHI-BFS-block`), a coarse-grain (32 threads) CC code (`PHI-DO`) that uses direction-optimized BFS idea [25], the hardware-vectorized code with $\mathcal{B} = 512$ (`PHI-SpMM-512`), the compiler-vectorized version using $\mathcal{B} = 8,192$ BFSs with (`PHI-SpMM-opt-comp-8192`) and without (`PHI-SpMM-comp-8192`) the modifications described at the end of Section 3.4.

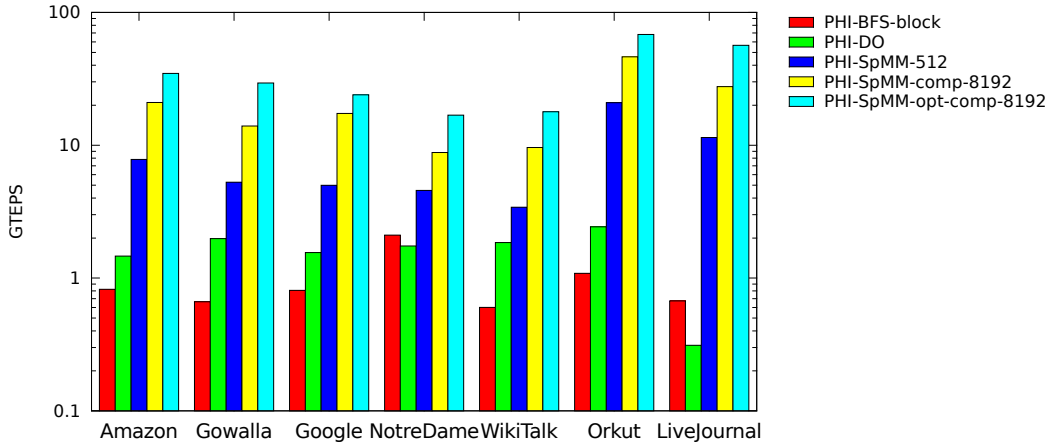


Figure 3.11: Performance of the configurations on Xeon Phi.

The first two methods do not use the proposed densification techniques and obtain low performance: the performance of `PHI-BFS-block` ranges from 600 MTEPS to 2.1 GTEPS, while `PHI-DO` sees its performance range from 311 MTEPS to 2.4 GTEPS. On the other hand, `PHI-SpMM-opt-comp-8192` is, on the average, 22.1 times faster than `PHI-DO` and its performance is between 16.8 GTEPS to 68.2 GTEPS. Also, the modifications to take shortcuts bring a 1.75 factor of improvement on the average.

As the other SpMM-based codes, `PHI-SpMM-opt-comp-8192` is composed of three phases: `Init`, `SpMM` and `Update`. The relative proportions of the execution times of these phases depend on the structure of the graph as shown in Figure 3.12. For all the graphs, the time required for `Init` is smaller compared to other phases. However, the relative time for `SpMM` and `Update` drastically changes with the graph, e.g., see `NotreDame` and `Orkut`. Figure 3.13 shows how the time of the `SpMM` phase and the

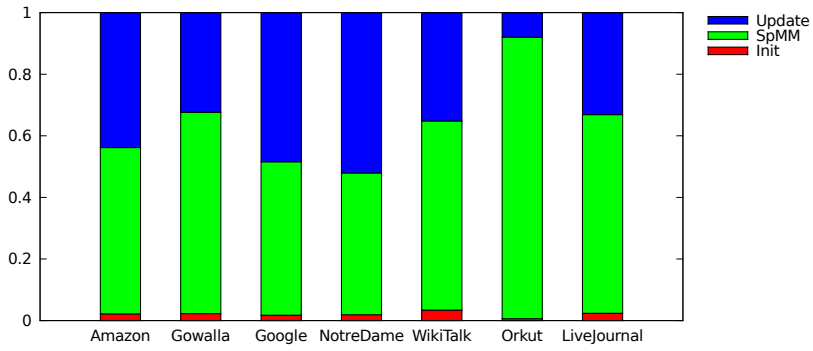


Figure 3.12: Proportion of each section of the execution time of `PHI-SpMM-comp-opt-8192`

`Update` phase vary with the iterations among the levels of the BFSs and how many vertices are actually processed in each of these phases. One can see that the time spent in the `SpMM` phase is fairly well correlated to the number of vertices processed in this phase (thanks to the modifications at the end of Section 3.4). A similar pattern exists on the `Update` phase. The non-modified version, which is not shown here, has much flatter execution times for these two phases; the amount of improvement provided by the modifications depends on the distribution of these skipped vertices which varies from one graph to the other.

SpMM-based closeness centrality on GPU

The performance of the SpMM-based approach on the GPU depends on how many traversals are performed simultaneously, the data type used, and how many threads/warps are used per vertex. Overall, as in Xeon Phi experiments, when $\mathcal{B}$ increases so does the performance. In addition to 64-bit integers, we also tried using

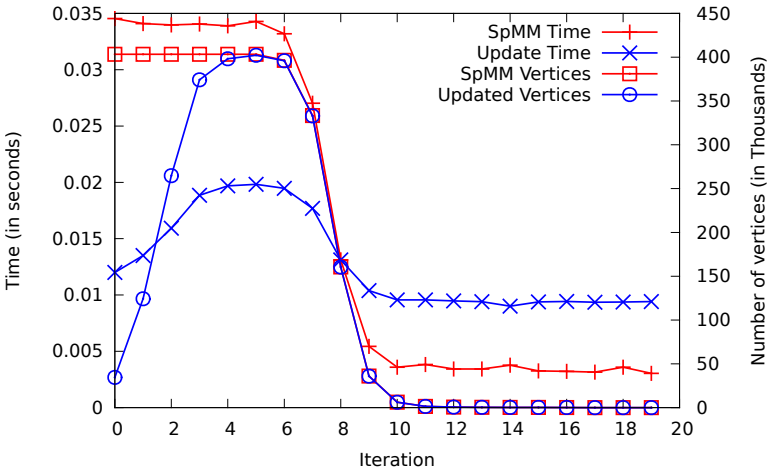


Figure 3.13: Time break-down per iteration and number of updated vertices for the `Amazon` graph. The variation of the time is explained by the number of vertices processed during those phase.

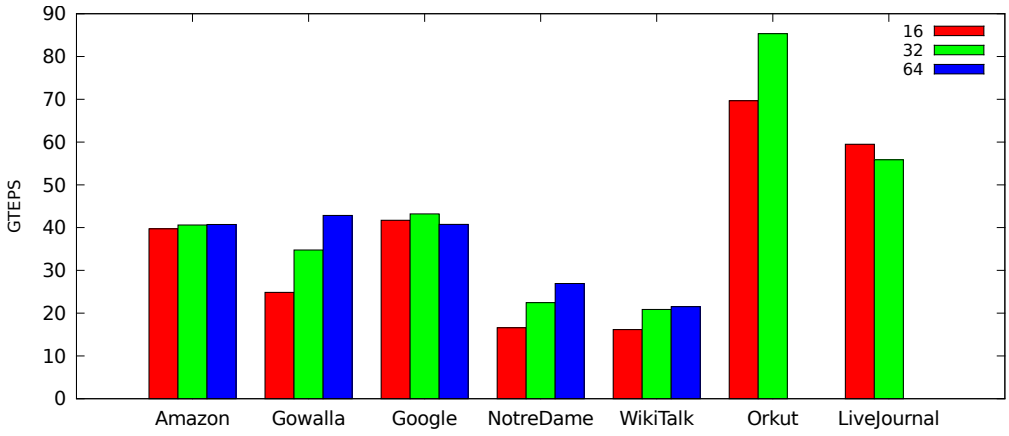


Figure 3.14: Impact on the number of threads per vertex on the performance of GPU-SpMM.

32-bit ones in our preliminary experiments which performed almost always worse than the 64-bit version. Therefore, Figure 3.14 uses the 64-bit version and shows the performance of the GPU-based algorithm using different number of threads/warps per vertex. Since the NVIDIA Tesla K20 has a relatively small memory, which is 6GB, $\mathcal{B}$, the maximum number of simultaneous BFSs, is set to $2,048$ for `Orkut` and `LiveJournal`, $4,096$ for `WikiTalk` and $8,192$ for `Amazon`, `Gowalla`, `Google`, and `NotreDame`. The figure does not contain 2-warp per vertex (64 threads) configuration for `Orkut` and `LiveJournal` since there are only $2,048/64 = 32$ integers due to the memory restriction. Hence, even we assign 2-warps per vertex, one of the warps will stay idle.

The performance of the SpMM-based approach varies with the number of threads per vertex. The performance usually increases when we use a single warp (32 threads) instead of a half-warp (16 threads) per vertex (except `LiveJournal`). This is expected since although the memory accesses of the threads in each half are coordinated, a half still need to wait the other especially when the lengths of the adjacency lists assigned to these halves significantly differ. Using two warps (64 threads) also increases the performance but less frequently. For example, the increase for `Amazon` and `WikiTalk`, are not significant, and there is a performance decrease for `Google`. We will use 32 threads per each vertex in the rest of the text while presenting the performance of GPU-based implementation.

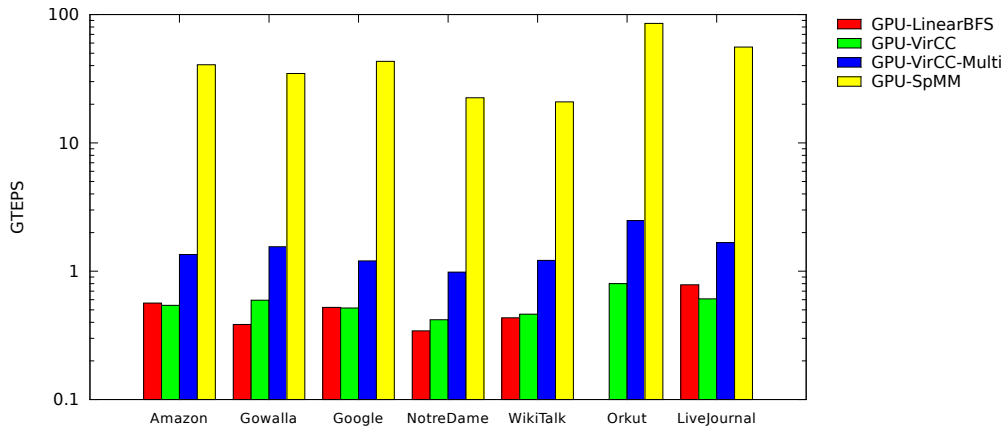


Figure 3.15: Comparison of GPU-based CC algorithms.

We compare the performance of `GPU-SpMM` with multiple baselines from the literature in Figure 3.15. `GPU-LinearBFS` is the linear-time, fine-grain, parallel BFS implementation proposed for GPU [122]. `GPU-VirCC` is a direct adaptation of `GPU-VirBC` (from [148]), and `GPU-VirCC-Multi` is a direct adaptation of `GPU-VirBC-Multi` (from Section 3.3.1) to closeness centrality. Similar to BC experiments, we used $\Delta = 8$ for virtualization. With the help of simultaneous BFSs, `GPU-VirCC-Multi` performs better than the single-BFS variant `GPU-VirCC`. However, the `GPU-SpMM` algorithm performs one order of magnitude faster than the rest thanks to vectorization and a more compact formulation.

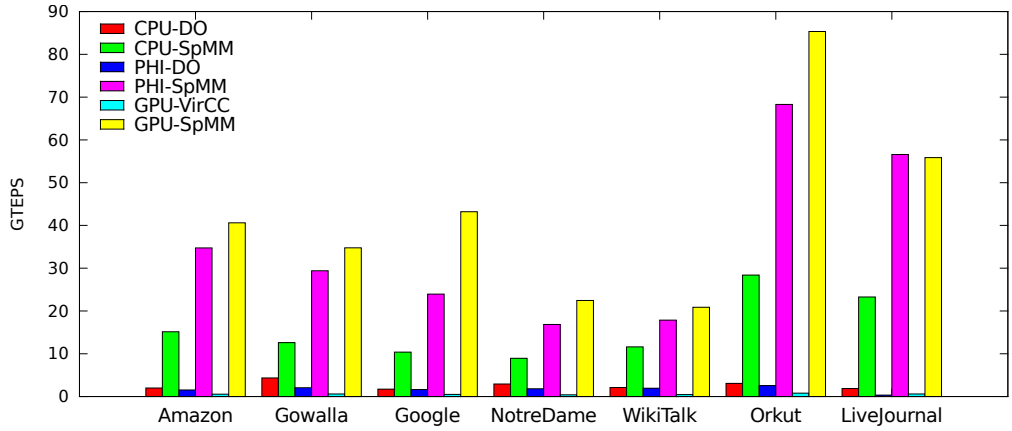


Figure 3.16: Vectorization works: `CPU-SpMM` is the compiler-vectorized implementation executed on CPU (32 threads) with $\mathcal{B} = 4,096$. `PHI-SpMM` is the corresponding Xeon Phi variant with $\mathcal{B} = 8,192$. For the GPU-based implementation, the maximum possible $\mathcal{B}$ value is used for each graph, and a vertex is assigned to a warp (32 threads).

Summary of the closeness centrality experiments

In Figure 3.16, we present the performance of the SpMM-based CC implementation on all the three architectures with the best non-vectorized algorithm from the literature and the best vectorized algorithm described in this work. On CPU and Xeon Phi, $4,096$ and $8,192$ simultaneous BFSs, respectively, are used. On GPU, the maximum possible simultaneous BFSs is used for each graph as described above. For the non-vectorized variants, the direction optimized CC variant performs the best on CPU and Xeon Phi, while the `GPU-VirCC` algorithm with simultaneous BFSs performs best on the GPU. On average, the vectorized algorithm is 5.9 times faster than

the non-vectorized one on CPU, 21.0 times faster on Intel Xeon Phi, and 70.4 times faster on NVIDIA Tesla K20c than the best existing ones.

3.5 Summary and Future Work

In this work, we proposed new algorithms and parallelization techniques to make betweenness and closeness centrality computations faster on commonly available cutting edge hardware. There are two traditional ways to execute centrality computations in parallel. Either each thread traverses the graph from a single source, or all the threads collaboratively traverse the graph from a unique source. We deviated from the traditional approaches by using all the threads in the system to collaboratively traverse the graph from many sources simultaneously. This scheme makes the computations more regular and allows a better utilization of modern computing devices. The experimental evaluation of the proposed algorithms shows that significant improvements can be obtained over the best known algorithms for centrality computation on the same device, without using an additional hardware: an improvement of a factor 5.9 on CPU architectures, 70.4 on GPU architectures and 21.0 on Intel Xeon Phi.

The techniques can be applied to these architectures at the same time. Hence, they are suitable for heterogeneous computing which is straightforward for centrality computations as we have shown in [148]. Furthermore, the proposed approach is also suitable to compute approximate centrality values. In the future, we want to analyze the impact of vectorization in the streaming setting for dynamic networks. But more

importantly, we want to investigate whether other common graph computations can be regularized.

Chapter 4: Incremental Closeness Centrality Algorithms and Parallelization

Dynamic nature of today's networks requires new algorithms. Maintaining the exact centrality scores is a challenging problem which has been studied in the literature [76, 104, 151]. The problem can also arise for applications involving static networks such as the power grid contingency analysis and robustness evaluation of a network. The findings of such analyses and evaluations can be very useful to be prepared and take proactive measures if there is a natural risk or a possible adversarial attack that can yield undesirable changes on the network topology in the future. In some applications, similarly, one might be interested in trying to find the minimal topology modifications on a network to set the centrality scores in a controlled manner.

4.1 Introduction

In this chapter, we focus on incremental closeness centrality computation problem. We propose incremental algorithms which efficiently update the closeness centralities upon edge insertions and deletions. Compared with the existing algorithms, our

algorithms have a low-memory footprint which makes them practical and applicable to very large graphs. For random edge insertions/deletions to the Wikipedia users' communication graph, we reduced the centrality (re)computation time from 2 days to 16 minutes. And for the real-life temporal DBLP coauthorship network, we reduced the time from 1.3 days to 4.2 minutes.

Furthermore, we propose the first distributed-memory framework STREAMER for the incremental centrality computation problem which employs a pipelined parallelism to achieve computation-computation and computation-communication overlap. In our experiments, the worker nodes we used in the experiments have 8 cores. In addition to the distributed-memory parallelization, we also leverage the shared-memory parallelization and take NUMA effects into account. The framework appears to scale linearly: when 63 worker nodes (8 cores/node) are used, for the networks `amazon0601` and `web-Google`, STREAMER obtains 456 and 497 speedups, respectively, compared to a single worker node-single thread execution. The STREAMER framework is modular which makes it easily extendable. When the number of used nodes increases, the computation inevitably reaches a bottleneck on the extremities of the analysis pipeline which are not parallel. We show how the computation can be made parallel by leveraging the modularity of dataflow middleware. Furthermore, using an SpMM-based BFS formulation, we significantly improved the incremental CC computation performance and show that the dataflow programming model makes STREAMER highly

modular and easy to enhance with novel algorithmic techniques. Those additional techniques provide an improvement of a factor between 2.2 to 9.3 times.

Background on closeness centrality computation can be found in Section 2.2 of Chapter 2. Rest of this chapter is organized as follows: Our algorithms are explained in detail in Section 4.2. STREAMER is explained in Section 4.4. An experimental analysis is given in Section 4.5. Related works are given in Section 4.6 and Section 4.7 concludes the chapter.

4.2 Maintaining Centrality

Many real-life networks are scale free. The diameters of these networks grow proportional to the logarithm of the number of nodes. That is, even with hundreds of millions of vertices, the diameter is small, and when the graph is modified with minor updates, it tends to stay small. Combining this with the power-law degree distribution of scale-free networks, we obtain the spike-shaped shortest-distance distribution as shown in Figure 4.1. We use *work filtering with level differences* and *utilization of special vertices* to exploit these observations and reduce the centrality computation time. In addition, we apply *SSSP hybridization* to speedup each SSSP computation.

4.2.1 Work Filtering with Level Differences

For efficient maintenance of the closeness centrality values in case of an edge insertion/deletion, we propose a *work filter* which reduces the number of SSSPs in Algorithm 1 and the cost of each SSSP by utilizing the level differences.

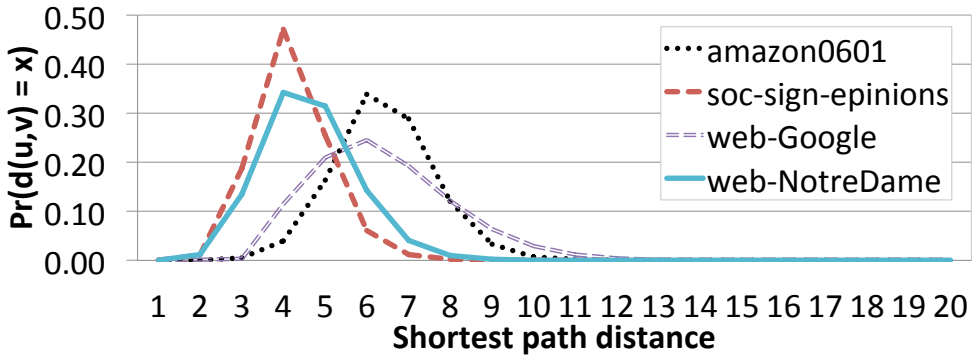


Figure 4.1: The probability of the distance between two (connected) vertices is equal to x for four social and web networks.

Level-based filtering detects the unnecessary updates and filter them out. Let $G = (V, E)$ be the current graph and uv be an edge to be inserted to G. Let $G' = (V, E \cup uv)$ be the updated graph. The centrality definition in (2.2.1) implies that for a vertex $s \in V$, if $\texttt{dst}_G(s,t) = \texttt{dst}_{G'}(s,t)$ for all $t \in V$ then $\texttt{cc}[s] = \texttt{cc}'[s]$. The following theorem is used to detect such vertices and filter their SSSPs.

Theorem 3. *Let $G = (V, E)$ be a graph and u and v be two vertices in V s.t. $uv \notin E$. Let $G' = (V, E \cup uv)$. Then $\texttt{cc}[s] = \texttt{cc}'[s]$ if and only if $|\texttt{dst}_G(s,u) - \texttt{dst}_G(s,v)| \leq 1$.*

Proof. If s is disconnected from u and v, uv's insertion will not change $\texttt{cc}[s]$. Hence, $\texttt{cc}[s] = \texttt{cc}'[s]$. If s is only connected to one of u and v in G the difference $|\texttt{dst}_G(s,u) - \texttt{dst}_G(s,v)|$ is ∞, and $\texttt{cc}[s]$ needs to be updated by using the new, larger connected component containing s. When s is connected to both u and v in G, we investigate the edge insertion in three cases as shown in Figure 4.2:

Case 1: $\texttt{dst}_G(s,u) = \texttt{dst}_G(s,v)$: Assume that the path $s \overset{P}{\rightsquigarrow} u$–$v \overset{P'}{\rightsquigarrow} t$ is a shortest $s \rightsquigarrow t$ path in G' containing uv. Since $\texttt{dst}_G(s,u) = \texttt{dst}_G(s,v)$, there exists a shorter path $s \overset{P''}{\rightsquigarrow} v \overset{P'}{\rightsquigarrow} t$ with one less edge. Hence, $\forall t \in V$, $\texttt{dst}_G(s,t) = \texttt{dst}_{G'}(s,t)$.

Case 2: $|\texttt{dst}_G(s,u) - \texttt{dst}_G(s,v)| = 1$: Let $\texttt{dst}_G(s,u) < \texttt{dst}_G(s,v)$. Assume that $s \overset{P}{\rightsquigarrow} u$–$v \overset{P'}{\rightsquigarrow} t$ is a shortest path in G' containing uv. Since $\texttt{dst}_G(s,v) = \texttt{dst}_G(s,u)+1$,

there exists another path $s \overset{P''}{\rightsquigarrow} v \overset{P'}{\rightsquigarrow} t$ with the same length. Hence, $\forall t \in V$, $\texttt{dst}_G(s,t) = \texttt{dst}_{G'}(s,t)$.

Case 3: $|\texttt{dst}_G(s,u) - \texttt{dst}_G(s,v)| > 1$: Let $\texttt{dst}_G(s,u) < \texttt{dst}_G(s,v)$. The path $s \rightsquigarrow u$–v in G' is shorter than the shortest $s \rightsquigarrow v$ path in G since $\texttt{dst}_G(s,v) > \texttt{dst}_G(s,u) + 1$. Hence, $\forall t \in V \setminus \{v\}, \texttt{dst}_{G'}(s,t) \leq \texttt{dst}_G(s,t)$ and $\texttt{dst}_{G'}(s,v) < \texttt{dst}_G(s,v)$, i.e., an update on $\texttt{cc}[s]$ is necessary. □

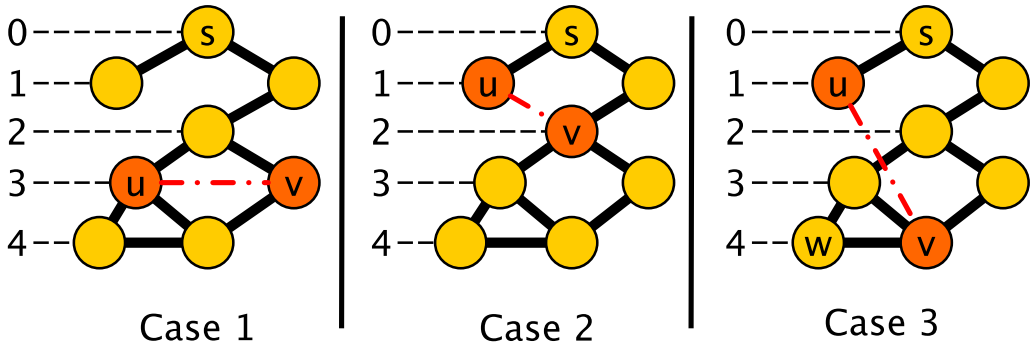


Figure 4.2: Three cases of edge insertion: when an edge uv is inserted to the graph G, for each vertex s, one of them is true: (1) $\texttt{dst}_G(s,u) = \texttt{dst}_G(s,v)$, (2) $|\texttt{dst}_G(s,u) - \texttt{dst}_G(s,v)| = 1$, and (3) $|\texttt{dst}_G(s,u) - \texttt{dst}_G(s,v)| > 1$.

Although Theorem 3 yields to a filter only in case of edge insertions, the following corollary which is used for edge deletion easily follows.

Corollary 1. *Let $G = (V,E)$ be a graph and u and v be two vertices in V s.t. $uv \in E$. Let $G' = (V, E \setminus \{uv\})$. Then $\texttt{cc}[s] = \texttt{cc}'[s]$ if and only if $|\texttt{dst}_{G'}(s,u) - \texttt{dst}_{G'}(s,v)| \leq 1$.*

With this corollary, the work filter can be implemented for both edge insertions and deletions. The pseudocode of the update algorithm in case of an edge insertion is given in Algorithm 10. When an edge uv is inserted/deleted, to employ the filter,

we first compute the distances from u and v to all other vertices. And, we filter the vertices satisfying the statement of Theorem 3.

Algorithm 10: Simple work filtering

Data: $G = (V, E)$, $\texttt{cc}[.]$, uv
Output: $\texttt{cc}'[.]$

```
1 G' ← (V, E ∪ {uv})
2 dstu[.] ← SSSP(G, u) ▷ distances from u in G
3 dstv[.] ← SSSP(G, v) ▷ distances from v in G
4 for each s ∈ V do
5     if |dstu[s] − dstv[s]| ≤ 1 then
6         cc'[s] = cc[s]
7     else
          ▷ use the computation in Algorithm 1 with G'
8 return cc'[.]
```

In theory, filtering by levels can reduce the update time significantly. However, in practice, its effectiveness depends on the underlying structure of G. Many real-life networks have been repeatedly shown to possess unique characteristics such as a small diameter and a power-law degree distribution [119]. And the spread of information is extremely fast [52, 53]. The proposed filter exploits one of these characteristics for efficient closeness centrality updates: the distribution of shortest-path lengths. Its efficiency is based on the phenomenon shown in Figure 4.1 for a set of graphs used in our experiments: the probability distribution function for a shortest-path length being equal to x is unimodular and spike-shaped for many real-life networks. This is the outcome of the short diameter and power-law degree distribution. On the other

hand, for some spatial networks such as road networks, there are no sharp peaks and the shortest-path distances are distributed in a more uniform way. The work filter we propose here prefers the former.

4.2.2 Utilization of Special Vertices

We exploit some special vertices to speedup the incremental closeness centrality computation further. We leverage the articulation vertices and identical vertices in networks. Although it has been previously shown that articulation vertices in real social networks are limited and yield an unbalanced shattering [152], we present the related techniques here to give a complete view.

Filtering with biconnected components

Our filter can be assisted by maintaining a biconnected component decomposition (BCD) of $G = (V, E)$. A BCD is a partitioning Π of E where $\Pi(e)$ is the component of each edge $e \in E$.

When uv is inserted to G and $G' = (V, E' = E \cup \{uv\})$ is obtained, we check if

$$\{\Pi(uw) : w \in \Gamma_G(u)\} \cap \{\Pi(vw) : w \in \Gamma_G(v)\}$$

is empty or not: if the intersection is not empty, there will be only one element in it, *cid*, which is the id of the biconnected component of G' containing uv (otherwise Π is not a valid BCD). In this case, $\Pi'(e)$ is set to $\Pi(e)$ for all $e \in E$ and $\Pi'(uv)$ is set to *cid*. If there is no biconnected component containing both u and v , i.e., if the

intersection above is empty, we construct Π' from scratch and set $cid = \Pi'(uv)$. Π can be computed in linear, $\mathcal{O}(m+n)$ time [85]. Hence, the cost of BCD maintenance is negligible compared to the cost of updating closeness centrality.

Filtering with identical vertices

Our preliminary analyses show that real-life networks can contain a significant amount of *identical* vertices with the same/a similar neighborhood structure. We investigate two types of identical vertices.

Definition 1. *In a graph G, two vertices u and v are* type-I-identical if and only if $\Gamma_G(u) = \Gamma_G(v)$.

Definition 2. *In a graph G, two vertices u and v are* type-II-identical if and only if $\{u\} \cup \Gamma_G(u) = \{v\} \cup \Gamma_G(v)$.

Both types form an equivalence class relation since they are reflexive, symmetric, and transitive. Hence, all the classes they form are disjoint.

Let $u, v \in V$ be two identical vertices. One can see that for any vertex $w \in V \setminus \{u, v\}$, $\texttt{dst}_G(u, w) = \texttt{dst}_G(v, w)$. Then the following is true.

Corollary 2. *Let $\mathcal{I} \subseteq V$ be a vertex-class containing type-I or type-II identical vertices. Then the closeness centrality values of all the vertices in $\mathcal{I}$ are equal.*

To construct these equivalance classes for the initial graph, we first use a hash function to map each vertex neighborhood to an integer:

$$hash_I[u] = \sum_{v \in \Gamma_G(u)} v.$$

We then sort the vertices with respect to their hash values and construct the type-I vertex-classes by eliminating false positives due to collisions on the hash function.

A similar process is applied to detect type-II vertex classes. The complexity of this initial construction is $\mathcal{O}(n \log n + m)$ assuming the number of collisions is small and hence, false-positive detection cost is negligible.

Maintaining the equivalance classes in case of edge insertions and deletions is easy: For example, when uv is added to G, we first subtract u and v from their classes and insert them to new ones (or leave them as singletons if none of the vertices are now identical with them). The cost of this maintenance is $\mathcal{O}(n + m)$.

While updating the closeness centrality values of the vertices in V, we execute an SSSP for at most one vertex from each identical-vertex class. For the rest of the vertices, we use the same closeness centrality value. The improvement is straightforward and the modifications are minor. For brevity, we do not give the pseudocode.

4.2.3 SSSP Hybridization

The spike-shaped distribution given in Figure 4.1 can also be exploited for SSSP hybridization. Consider the execution of Algorithm 1: while executing an SSSP with source s, for each vertex pair $\{u, v\}$, u is processed before v if and only if $d_G(s, u) < d_G(s, v)$. That is, Algorithm 1 consecutively uses the vertices with distance k to find the vertices with distance $k + 1$. Hence, it visits the vertices in a *top-down* manner. SSSP can also be performed in a a *bottom-up* manner. That is to say, after all distance (level) k vertices are found, the vertices whose levels are unknown can be processed to see if they have a neighbor at level k. The top-down variant is expected

to be much cheaper for small k values. However, it can be more expensive for the upper levels where there are much less unprocessed vertices remaining.

Following the idea of Beamer et al. [25], we hybridize the SSSPs. While processing the nodes at an SSSP level, we simply compare the number of edges need to be processed for each variant and choose the cheaper one.

4.2.4 Simultaneous source traversal

The performance of sparse kernels is mostly hindered by irregular memory accesses. The most famous example for sparse computation is the multiplication of a sparse matrix by a dense vector (SpMV). Several techniques, like register blocking [36, 175] and usage of different matrix storage formats [26, 111], are proposed to regularize the memory access pattern. However, multiplying a sparse matrix by multiple vectors is the most efficient and popular technique to regularize the memory access pattern. Once the multiple vectors are organized as a dense matrix, the problem becomes the multiplication of a sparse matrix by a dense matrix (SpMM). Each nonzero of the sparse matrix causes the multiplication of a single element of the vector in SpMV, and it results in the multiplications of as many consecutive elements of the dense matrix as its number of columns in SpMM.

Accommodating that idea for closeness centrality computation turns out to be concurrently computing the multiple sources at the same time. However, as opposed to SpMV, in which the vector is dense and therefore each non-zero induces exactly one multiplication, in BFS, not all the non-zeros will induce operations. That is

to say, a vertex in BFS may or may not be traversed depending on which level is currently being processed. Thus, the traditional queue-based implementation of BFS does not seem to be easily extendable to support concurrent BFSs (co-BFS) in a vector-friendly manner. We developed this method in [154, 156] and present here the main idea.

An SpMV-based formulation of closeness centrality

The idea is to convert to a simpler definition of level synchronous BFS: If one of the neighbor of v is part of level $\ell - 1$ and v is not part of any level $\ell' < \ell$, then vertex v is part of level ℓ. This formulation is used in parallel implementations of BFS on GPU [90, 135, 164], on shared memory systems [3] and on distributed memory systems [35].

The algorithm is better represented using binary variables. Let x_i^ℓ be the binary variable that is `true` if vertex i is part of the frontier at level ℓ for a BFS. The neighbors of level ℓ is represented by a vector $y^{\ell+1}$ computed by $y_k^{\ell+1} = \texttt{OR}_{j \in \Gamma(k)} x_j^\ell$. The next level is then computed with $x_i^{\ell+1} = y_i^{\ell+1}$ `AND not` $(\texttt{OR}_{\ell' \leq \ell} x_i^{\ell'})$. Using these variables, one can increase the farness of the source by ℓ if i is at level ℓ (i.e., if $x^\ell = 1$). One can remark that $y^{\ell+1}$ is the result of the "multiplication" of the adjacency matrix of the graph by x^ℓ in the (`OR`,`AND`) semi-ring.

An SpMM-based formulation of closeness centrality

It is easy to derive an algorithm from the formulation given above for closeness centrality computation that processes multiple sources concurrently. Instead of manipulating a single vector x and y where each element is a single bit, one can encode 32-bit vectors for 32 BFSs so that one *int* can encode the state of a single vertex across the 32 BFSs. The algorithm becomes quite efficient as it does not use more memory and process 32 BFS concurrently. All the operations become simple bit-wise `and`, `or` and `not`.

Theoretically, the asymptotic complexity changes when BFS is implemented using an SpMM approach. The complexity of the traditional queue-based BFS algorithm is $\mathcal{O}(|E|)$. If the adjacency matrix is stored row-wise, the SpMM-based implementation boils down to a bottom-up implementation of BFS which has a natural write access pattern. However, it becomes impossible to only traverse the relevant nonzero of the matrix and the complexity of the algorithm becomes $\mathcal{O}(|E| \times L)$, where L is the diameter of the graph. Social networks have small world properties which implies that their diameter is low and we do not feel that this asymptotic factor of L will hinder performance.

Moreover, multiple BFSs are performed concurrently (here 32) which can recoup for the loss. In [154, 156], the algorithm computes the impact of the sources on all the vertices of the graph. What we presented in this section does the reverse and compute the impact of all the vertices of the graph on the sources. Despite

worse asymptotic complexity the performance of co-BFS outperforms traditional BFS approach [154, 156]. Moreover, such algorithm is compatible with the decomposition of the graph in biconnected components [152] which can lead to further improvement. Because this algorithm computes the farness of the sources, it can be used to compute centrality incrementally.

4.3 DataCutter

STREAMER employs *DataCutter* [27], our in-house dataflow programming framework for distributed memory systems. In DataCutter, the computations are carried by independent computing elements, called *filters*, that have different responsibilities and operate on data passing through them. DataCutter follows the component-based programming paradigm which has been used to describe and implement complex applications [79, 80, 81, 153] by way of components - distinct tasks with well-defined interfaces. This is also known as the filter-stream programming model [27] (a specific implementation of the dataflow programming model). A *stream* denotes a unidirectional data flow from some filters (i.e., the producers) to others (i.e., the consumers). Data flows along these *streams* in untyped *databuffers* so as to minimize various system overheads. A *layout* is a filter ontology which describes the set of application tasks, streams, and the connections required for the computation. By describing these components and the explicit data connections between them, the applications are decomposed along natural task boundaries according to the application domain. Therefore, the component-based application design is an intuitive process

with explicit demarcation of task responsibilities. Furthermore, the communication patterns are also explicit; each component includes its input data requirements and outputs in its description.

Applications composed of a number of individual tasks can be executed on parallel and distributed computing resources and gain extra performance over those run on strictly sequential machines. This is achieved by specifying a *placement* which is an instance of a *layout* with a mapping of the filters onto physical processors. There are three main advantages of this scheme: first, it exposes an abstract representation of the application which is decoupled from its practical implementation. Second, the coarse-grain dataflow programming model allows *replicated parallelism* by instantiating a given filter multiple times so that the work can be distributed among the instances to improve the parallelism of the application and the system's performance. And third, the execution is pipelined, allowing multiple filters to compute simultaneously on different iterations of the work. This *pipelined parallelism* is very useful to achieve overlapping of communication and computation.

Additionally, provided the interfaces exposed by a task to the rest of the application, different implementations of tasks, possibly on different processor architectures can co-exist in the same application deployment, allowing developers to take full advantage of modern, heterogeneous supercomputers. Figure 4.3 shows an example filter-stream layout and placement. In this work, we used both distributed- and shared-memory architectures. However, thanks to filter-stream programming model,

many-core systems such as GPUs and accelerators can also be used easily and efficiently if desired [81].

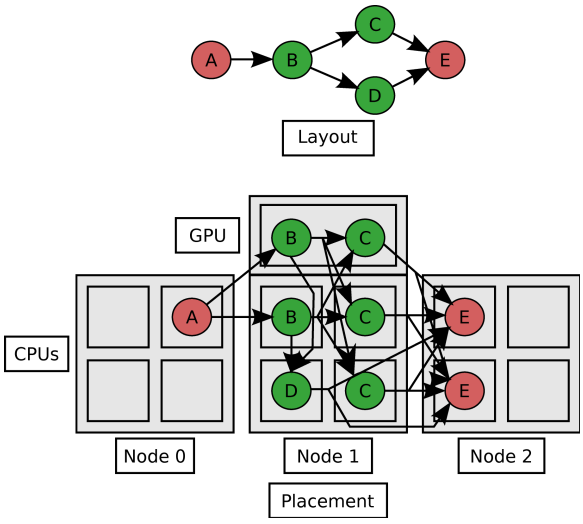


Figure 4.3: A toy filter-stream application layout and its placement.

As mentioned above, one of the DataCutter's strengths is that it enables pipelined parallelism, where multiple stages of the pipeline (such as A and B in the layout in Figure 4.3) can be executed simultaneously, and replicated parallelism can be used at the same time if some computation is stateless (such as filter B in the same figure). DataCutter makes all this parallelism is possible by mapping each placed filter to a POSIX thread of the execution platform.

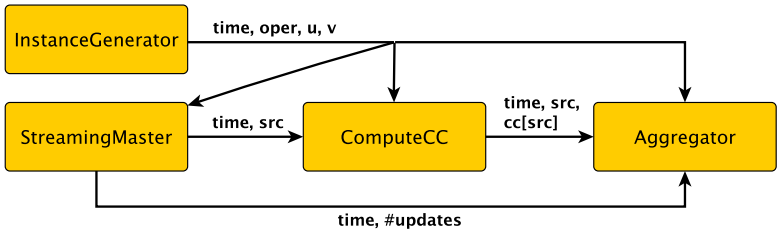


Figure 4.4: Layout of STREAMER.

4.4 STREAMER

STREAMER is implemented in the DataCutter framework. We propose to use the four-filter layout shown in Figure 4.4. *InstanceGenerator* is responsible for sending the updates to all the other components. *StreamingMaster* does the work filtering for each update, and generates the workload for following components. *ComputeCC* component executes the real work and computes the updated CC scores for each incoming update. *Aggregator* does the necessary adjustments related to identical vertex sets and biconnected component decomposition. While computing the CC scores, the main portion of the computation comes from performing SSSPs for the vertices whose scores need to be updated. If there are many updates (we use the term "update" to refer to the SSSP operation which updates the CC score of a vertex), that part of the computation should occupy most of the machine. A typical synchronous decomposition of the application makes the work filtering of a streaming event (handling a single edge change) wait for the completion of all the work incurred

by a previous streaming event. Since the worker nodes will wait for the work filtering to be completed, there can be a large waste of resources. We argue that the pipelined parallelism should be used to overlap the process of filtering the work and computing the updates on the graph. In this section, we explain each component in detail and define their responsibilities.

The first filter is the *InstanceGenerator* which first sends the initial graph to all the other filters. It then sends the streaming events as 4-tuples $(t, oper, u, v)$ to indicate that edge uv has been either added or removed (specified by *oper*) at a given time t. (In the following, we only explain the system for edge insertion, but it is essentially the same for an edge removal.) In a real world application, this filter would be listening on the network or on a database trigger for topology modifications; but in our experiments, all the necessary information is read from a file.

StreamingMaster is responsible for the work filtering after each network modification. Upon inserting uv at time t, it first computes the shortest distances from u and v to all other vertices at time $t-1$. Then, it adds the edge uv into its local copy of the graph and updates the identical vertex sets. It partitions the edges of the graph to its biconnected components by using the algorithm in [85] and finds the component containing uv. For each vertex $s \in V$, it decides whether its CC score needs to be recomputed by checking the following conditions: (1) $\texttt{dst}(s, u)$ and $\texttt{dst}(s, v)$ differ by at least 2 units at time $t-1$, (2) s is adjacent to an edge which is also in uv's

biconnected component, (3) s is the representative of its identical vertex set. *StreamingMaster* then informs the *Aggregator* about the number of updates it will receive for time t. Finally, it sends the list of SSSP requests to the *ComputeCC* filter, i.e., the corresponding source vertex ids whose CC scores need to be updated.

ComputeCC performs the real work and computes the new CC scores after each graph modification. It waits for work from *StreamingMaster*, and when it receives a CC update request under the form of a 2-tuple (t, s) (update time and source vertex id), *ComputeCC* advances its local graph representation to time t by using the appropriate updates from *InstanceGenerator*. If there is a change on the local graph, the biconnected component of uv is extracted, and a concise information of the graph structure and the set of articulation vertices are updated (as described in [151]). Finally, the exact CC score $\texttt{cc}[s]$ at time t is computed and sent to the *Aggregator* as a 3-tuple $(t, s, \texttt{cc}[s])$. *ComputeCC* can be replicated to fill up the whole distributed memory machine without any problem: as long as a replica reads the update requests in the order of non-decreasing time units, it is able compute the correct CC scores.

The *Aggregator* filter gets the graph at a time t from *InstanceGenerator*. Then, it obtains the number of updates for that time from *StreamingMaster*. It computes the identical vertex sets as well as the BCD. It gets the updated CC scores from *ComputeCC*. Due to the pipelined parallelism used in the system and the replicated parallelism of *ComputeCC*, it is possible that updates from a later time can be received; STREAMER stores them in a backlog for future processing. When a $(t, s, \texttt{cc}[s])$

tuple is processed, the CC score of s is updated. If s is the representative of an identical vertex set, the CC scores of all the vertices in the same set are updated as well. If s is an articulation point, then the CC scores of the vertices which are represented by s (and are not in the biconnected component of uv) are updated as well, by using the difference in the CC score of s between time t and $t-1$. Since *Aggregator* needs to know the CC scores at time $t-1$ to compute the centrality scores at time t, the system must be bootstrapped: the system computes explicitly all the centrality scores of the vertices for time $t = 0$.

4.4.1 Exploiting the shared memory architecture

The main portion of the execution time is spent by the *ComputeCC* filter. Therefore, it is important to replicate this filter as much as possible. Each replica of the filter will end up maintaining its own graph structure and computing its own BCD. Modern clusters are hierarchical and composed of distributed memory nodes where each node contains multiple processors featuring multiple cores that share the same memory space. For instance, the nodes used in our experiments are equipped with two processors, each having 4 cores.

It is a waste of computational power to recompute the data structure on each core. But it is also a waste of memory. Indeed, the cores of a processor typically share a common last level of cache and using the same memory space for all the cores in a processor might improve the cache utilization. We propose to split the *ComputeCC* filter in two separate filters which is transparent to the rest of the system thanks to

DataCutter being component-based. The *Preparator* filter constructs the decomposed graph for each Streaming Event it is responsible for. The *Executor* filter performs the real work on the decomposed graph. In DataCutter, the filters running on the same physical node act run in separate pthreads within the same MPI process making sharing the memory as easy as communicating pointers. The release of the memory associated with the decomposed graph is handled by atomically decreasing a counter by the *Executor*.

The decoupling of the graph management and the CC score computation allows to either creating a single graph representation on each distributed memory node or having a copy of the graph on each NUMA domain of the architecture. This is shown in Fig. 4.5.

4.4.2 Parallelizing *StreamingMaster*

When the number of cores used for *ComputeCC* increases, the relative importance of *ComputeCC* in the total runtime decreases. Theoretically, with an infinite number of cores for *ComputeCC*, the time required by it will drop to zero. In this case, the bottleneck of the application becomes the maximum rate at which *StreamingMaster* can generate updates request and the rate at which *Aggregator* can merge the computed results. To improve these rates, we replace them with a construct that allow parallel execution.

StreamingMaster is decomposed in three filters which are laid out according to Figure 4.6. Most of the work done by *StreamingMaster* is done by a filter (we still call

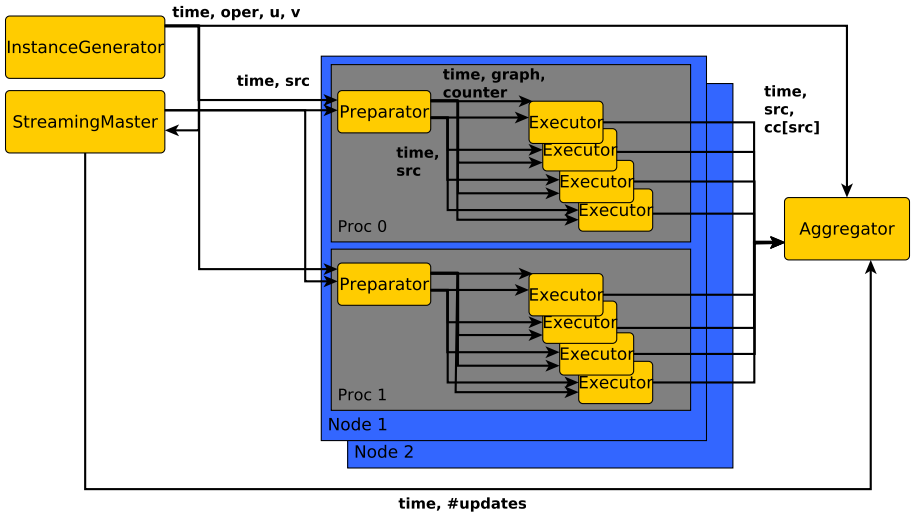


Figure 4.5: Placement of STREAMER using 2 worker nodes with 2 quad-core processors. (The node 2 is hidden). The remaining filters are on node 0.

it *StreamingMaster* for convenience) which supports replication. Each of the replica receives the list of edges it has to compute the filtering from a *WorkDistributor*. This *WorkDistributor* just listens the modifications on the graph and distribute the Streaming Events among different *StreamingMaster*s.

It is important that *ComputeCC* receives the update requests in non-decreasing order of streaming events. *StreamCoordinator* is responsible for enforcing that order. *StreamCoordinator* sits between the *StreamingMaster* and the *ComputeCC* (and the *Aggregator*) and relays messages to them. The *StreamCoordinator* tells *StreamingMaster* which streaming event is the next one. In other words, before outputting the

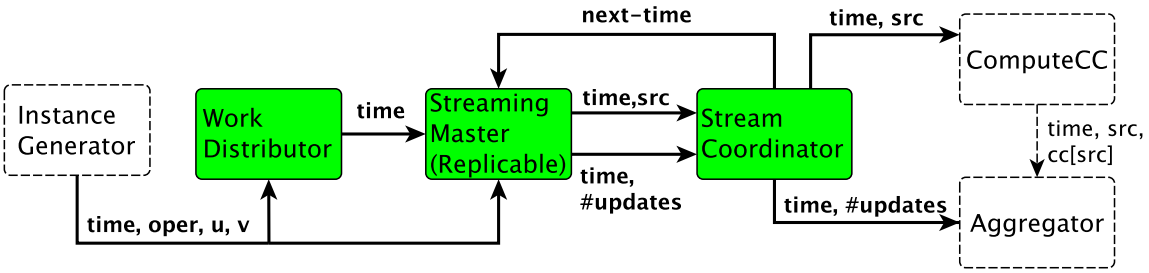


Figure 4.6: Replicating StreamingMaster for a better scaling when the number of processors is large.

list of updates (and metadata for the *Aggregator*), the *StreamingMaster* reads from the *StreamCoordinator* whether it is time to output.

4.4.3 Parallelizing *Aggregator*

One of the challenges in parallelizing the *Aggregator* is that there can be only one filter that actually stores the centrality values of the network. Fortunately, most of the computation time spent by the *Aggregator* is spent in preparing the network rather than in applying the updates. We modify the layout of the *Aggregator* to match that of Figure 4.7.

Therefore only a single filter, we will call *Aggregator* for the sake of simplicity, is responsible for applying the updates, and is only responsible for this. It takes three kinds of input: the updates on the graph itself, the information of how many updates will be applied for each streaming event and information on the graph (the graph itself, its biconnected decomposition and identical vertices).

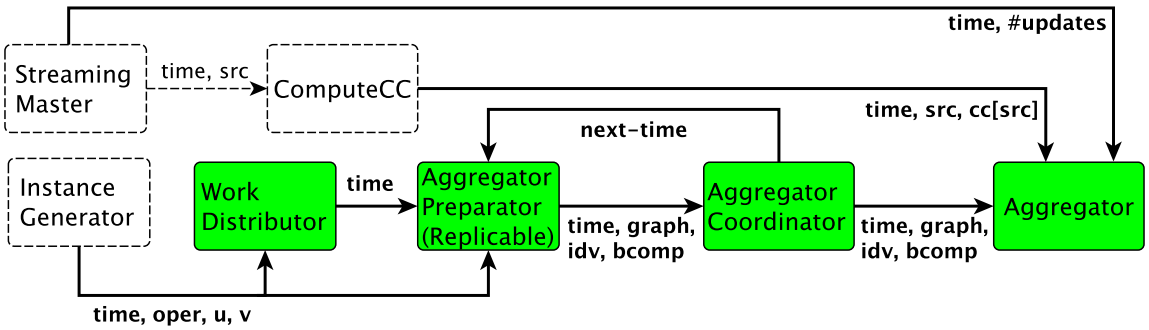


Figure 4.7: Replicating Aggregator for a better scaling when the number of processors is large.

The graph information is constructed by another filter called *AggregatorPreparator* which can be replicated. It listens to the Streaming Events and receive work assignments. It then computes the sets of identical vertices and the graph's biconnected component decomposition and send them through its downstream.

The work in the *AggregatorPreparator* is distributed in a way similar to the parallelization of the *StreamingMaster*. Also the graph information must reach the *Aggregator* in the order of the Streaming Event. An *AggregatorCoordinator* is used to regulate the order in which the graph information is sent. It behaves under the same principle as *StreamCoordinator*.

4.5 Experiments

We investigate the performance our algorithms and systems in two sections. First, we look at the impact of pure sequential algorithms on the performance, then we test

our STREAMER framework and compare its performance with respect to sequential algorithms.

4.5.1 Sequential Incremental Closeness Centrality

We implemented the algorithms in `C` and compiled with `gcc v4.6.2` with the optimization flags `-O2 -DNDEBUG`. The graphs are kept in the compressed row storage (CRS) format. The experiments are run in sequential on a computer with two Intel Xeon E5520 CPU clocked at 2.27GHz and equipped with 48GB of main memory.

For the experiments, we used 10 networks from the UFL Sparse Matrix Collection[2] and also extracted the coauthor network from the current set of DBLP papers. Properties of the graphs are summarized in Table 4.1. They are from different application areas, such as social (*hep-th*, *PGPgiantcompo*, *astro-ph*, *cond-mat-2005*, *soc-sign-epinions*, *loc-gowalla*, *amazon0601*, *wiki-Talk*, *DBLP-coauthor*), and web networks (*web-NotreDame*, *web-Google*). The graphs are listed by increasing number of edges and a distinction is made between small graphs (with less than 500K edges) and the large graphs (with more than 500K) edges.

Although the filtering techniques can reduce the update cost significantly in theory, their practical effectiveness depends on the underlying structure of G. Since the diameter of the social networks are small, the range of the shortest distances is small. Furthermore, the distribution of these distances is unimodal. When the distance with

[2] http://www.cise.ufl.edu/research/sparse/matrices/

Graph			Time (in sec.)		
name	$\|V\|$	$\|E\|$	Org.	Best	Speedup
hep-th	8.3K	15.7K	1.41	0.05	29.4
PGPgiantcompo	10.6K	24.3K	4.96	0.04	111.2
astro-ph	16.7K	121.2K	14.56	0.36	40.5
cond-mat-2005	40.4K	175.6K	77.90	2.87	27.2
				geometric mean	**43.5**
soc-sign-epinions	131K	711K	778	6.25	124.5
loc-gowalla	196K	950K	2,267	53.18	42.6
web-NotreDame	325K	1,090K	2,845	53.06	53.6
amazon0601	403K	2,443K	14,903	298	50.0
web-Google	875K	4,322K	65,306	824	79.2
wiki-Talk	2,394K	4,659K	175,450	922	190.1
DBLP-coauthor	1,236K	9,081K	115,919	251	460.8
				geometric mean	**99.8**

Table 4.1: The graphs used in the experiments. Column *Org.* shows the initial closeness computation time of CC and *Best* is the best update time we obtain in case of streaming data.

the peak (mode) is combined with the ones on its right and left, they cover a significant amount of the pairs (56% for *web-NotreDame*, 65% for *web-Google*, 79% for *amazon0601*, and 91% for *soc-sign-epinions*). We expect the filtering procedure to have a significant impact on social networks because of their structure. Besides, that specific structure is also important for the SSSP hybridization.

Handling topology modifications

To assess the effectiveness of our algorithms, we need to know when each edge is inserted to/deleted from the graph. Our datasets from the UFL collection do not have this information. To conduct our experiments on these datasets, we delete 1,000 edges from a graph chosen randomly in the following way: A vertex $u \in V$ is selected

randomly (uniformly), and a vertex $v \in \Gamma_G(u)$ is selected randomly (uniformly). Since we do not want to change the connectivity in the graph (having disconnected components can make our algorithms much faster and it will not be fair to CC), we discard uv if it is a bridge. If this is not the case we delete it from G and continue. We construct the initial graph by deleting these 1,000 edges. Each edge is then re-inserted one by one, and our algorithms are used to recompute the closeness centrality scores after each insertion.

In addition to the random insertion experiments, we also evaluated our algorithms on a real temporal dataset of the DBLP coauthor graph[3]. In this graph, there is an edge between two authors if they published a paper together. We used the publication dates as timestamps and constructed the initial graph with the papers published before January 1, 2013. We used the coauthorship edges of the later papers for edge insertions. Although we used insertions in our experiments, a deletion is a very similar process which should give comparable results.

In addition to CC, we configure our algorithms in four different ways: CC-B only uses BCD, CC-BL uses BCD and filtering with levels, CC-BLI uses all three work filtering techniques including identical vertices. And CC-BLIH uses all the techniques described in this work including the SSSP hybridization.

Table 4.2 presents the results of the experiments. The second column, CC, shows the time to run the full base algorithm for computing the closeness centrality values

[3] http://www.informatik.uni-trier.de/~ley/db/

Graph	Time (secs)					Speedups				Filter time (secs)
	CC	CC-B	CC-BL	CC-BLI	CC-BLIH	CC-B	CC-BL	CC-BLI	CC-BLIH	
hep-th	1.413	0.317	0.057	0.053	0.048	4.5	24.8	26.6	29.4	0.001
PGPgiantcompo	4.960	0.431	0.059	0.055	0.045	11.5	84.1	89.9	111.2	0.001
astro-ph	14.567	9.431	0.809	0.645	0.359	1.5	18.0	22.6	40.5	0.004
cond-mat-2005	77.903	39.049	5.618	4.687	2.865	2.0	13.9	16.6	27.2	0.010
Geometric mean	9.444	2.663	0.352	0.306	0.217	3.5	26.8	30.7	43.5	0.003
soc-sign-epinions	778.870	257.410	20.603	19.935	6.254	3.0	37.8	39.1	124.5	0.041
loc-gowalla	2,267.187	1,270.820	132.955	135.015	53.182	1.8	17.1	16.8	42.6	0.063
web-NotreDame	2,845.367	579.821	118.861	83.817	53.059	4.9	23.9	33.9	53.6	0.050
amazon0601	14,903.080	11,953.680	540.092	551.867	298.095	1.2	27.6	27.0	50.0	0.158
web-Google	65,306.600	22,034.460	2,457.660	1,701.249	824.417	3.0	26.6	38.4	79.2	0.267
wiki-Talk	175,450.720	25,701.710	2,513.041	2,123.096	922.828	6.8	69.8	82.6	190.1	0.491
DBLP-coauthor	115,919.518	18,501.147	288.269	251.557	252.647	6.2	402.1	460.8	458.8	0.530
Geometric mean	13,884.152	4,218.031	315.777	273.036	139.170	3.2	43.9	50.8	99.7	0.146

Table 4.2: Execution times in seconds of all the algorithms and speedups when compared with the basic closeness centrality algorithm CC. In the table CC-B is the variant which uses only BCDs, CC-BL uses BCDs and filtering with levels, CC-BLI uses all three work filtering techniques including identical vertices. And CC-BLIH uses all the techniques described in this work including SSSP hybridization.

on the original version of the graph. Columns 3–6 of the table present absolute runtimes (in seconds) of the centrality computation algorithms. The next four columns, 7–10, give the speedups achieved by each configuration. For instance, on the average, updating the closeness values by using CC-B on *PGPgiantcompo* is 11.5 times faster than running CC. Finally the last column gives the overhead of our algorithms per edge insertion, i.e., the time necessary to filter the source vertices and to maintain BCD and identical-vertex classes. Geometric means of these times and speedups are also given to provide a comparison across all the instances.

The times to compute the closeness values using CC on the small graphs range between 1 to 77 seconds. On large graphs, the times range from 13 minutes to 49 hours. Clearly, CC is not suitable for real-time network analysis and management based

on shortest paths and closeness centrality. When all the techniques are used (CC-BLIH), the time necessary to update the closeness centrality values of the small graphs drops below 3 seconds per edge insertion. The improvements range from a factor of 27.2 (*cond-mat-2005*) to 111.2 (*PGPgiantcompo*), with an average improvement of 43.5 across small instances and a factor of 42.6 (*loc-gowalla*) to 458.8 (*DBLP-coauthor*), on large graphs, with an average of 99.7. For all graphs, the time spent for overheads is below one second which indicates that the majority of the time is spent for SSSPs. Note that this part is pleasingly parallel since each SSSP is independent from each other. Hence, by combining the techniques proposed in this work with a straightforward parallelism, one can obtain a framework that can maintain the closeness centrality values within a dynamic network in real time.

The overall improvement obtained by the proposed algorithms is significant. The speedup obtained by using BCDs (CC-B) are 3.5 and 3.2 on the average for small and large graphs, respectively. The graphs *PGPgiantcompo*, and *wiki-Talk* benefits the most from BCDs (with speedups 11.5 and 6.8, respectively). Clearly using the biconnected component decomposition improves the update performance. However, filtering by level differences is the most efficient technique: CC-BL brings major improvements over CC-B. For all social networks, when CC-BL is compared with CC-B, the speedups range from 4.8 (*web-NotreDame*) to 64 (*DBLP-coauthor*). Overall, CC-BL brings a 7.61 times improvement on small graphs and a 13.44 times improvement on large graphs over CC.

For each added edge uv, let X be the random variable equal to $|\texttt{dst}_G(u,w) - \texttt{dst}_G(v,w)|$. By using 1,000 uv edges, we computed the probabilities of the three cases we investigated before and give them in Fig. 4.8. For each graph in the figure, the sum of the first two columns gives the ratio of the vertices not updated by CC-BL. For the networks in the figure, not even 20% of the vertices require an update ($Pr(X > 1)$). This explains the speedup achieved by filtering using level differences. Therefore, level filtering is more useful for the graphs having characteristics similar to small-world networks.

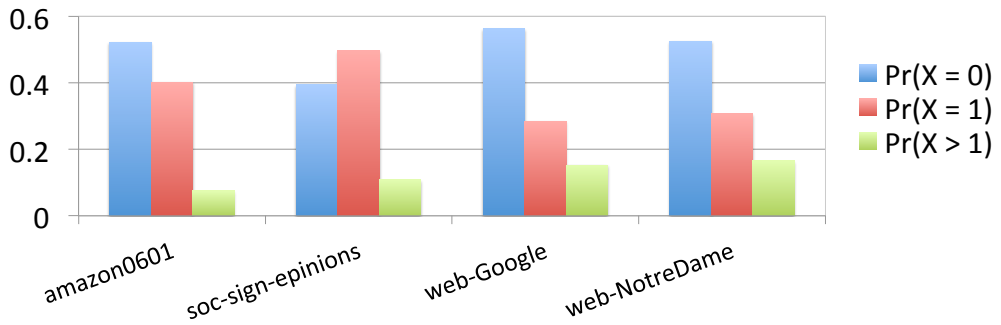


Figure 4.8: The bars show the distribution of random variable $X = |\texttt{dst}_G(u,w) - \texttt{dst}_G(v,w)|$ into three cases we investigated when an edge uv is added.

Filtering with identical vertices is not as useful as the other two techniques in the work filter. Overall, there is a 1.15 times improvement with CC-BLI on both small and large graphs compared to CC-BL. For some graphs, such as *web-NotreDame* and *web-Google*, improvements are much higher (30% and 31%, respectively).

The algorithm with the hybrid SSSP implementation, CC-BLIH, is faster than CC-BLI by a factor of 1.42 on small graphs and by a factor of 1.96 on large graphs. Although it seems to improve the performance for all graphs, in some few cases, the performance is not improved significantly. This can be attributed to incorrect decisions on SSSP variant to be used. Indeed, we did not benchmark the architecture to discover the proper parameter. CC-BLIH performs the best on social network graphs with an improvement ratio of 3.18 (*soc-sign-epinions*), 2.54 (*loc-gowalla*), and 2.30 (*wiki-Talk*).

All the previous results present the average single edge update time for 1,000 successively added edges. Hence, they do not say anything about the variance. Figure 4.9 shows the runtimes of CC-B and CC-BLIH per edge insertion for *web-NotreDame* in a sorted order. The runtime distribution of CC-B clearly has multiple modes. Either the runtime is lower than 100 milliseconds or it is around 700 seconds. We see here the benefit of BCD. According to the runtime distribution, about 59% of *web-NotreDame*'s vertices are inside small biconnected components. Hence, the time per edge insertion drops from 2,845 seconds to 700.

Indeed, the largest component only contains 41% of the vertices and 76% of the edges of the original graph. The decrease in the size of the components accounts for the gain of performance.

The impact of level filtering can also be seen on Figure 4.9. 60% of the edges in the main biconnected component do not change the closeness values of many vertices

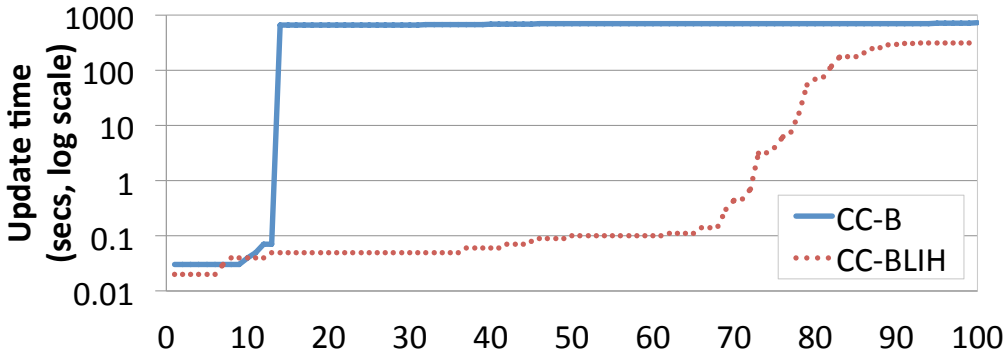


Figure 4.9: Sorted list of the runtimes per edge insertion for the first 100 added edges of *web-NotreDame*.

and the updates that are induced by their addition take less than 1 second. The remaining edges trigger more expensive updates upon insertion. Within these 30% expensive edge insertions, using identical vertices and SSSP hybridization provide a significant improvement (not shown in the figure).

Better Speedups on Real Temporal Data The best speedups are obtained on the DBLP coauthor network which uses real temporal data. Using CC-B, we reach 6.2 speedup w.r.t. CC, which is bigger than the average speedup on all networks. Main reason for this behavior is that 10% of the inserted edges are actually the new vertices joining to the network, i.e., authors with their first publication, and CC-B handles these edges quite fast. Applying CC-BL gives a 64.8 speedup over CC-B, which is drastically higher than all other graphs. Indeed, only 0.7% of the vertices require to run a SSSP algorithm when an edge is inserted on the DBLP network.

For the synthetic cases, this number is 12%. Overall, speedups obtained with real temporal data reach 460.8, i.e., 4.6 times greater than the average speedup on all graphs. Our algorithms appear to perform much better on real applications than on synthetic ones.

4.5.2 STREAMER

STREAMER runs on the *Owens* cluster in the Department of Biomedical Informatics at The Ohio State University. For the experiments, we used all the 64 computational nodes, each with dual Intel Xeon E5520 Quad-core CPUs (with 2-way Simultaneous Multithreading, and 8MB of L3 cache per processor), 48 GB of main memory. The nodes are interconnected with 20 Gbps InfiniBand. The algorithms were run on CentOS 6, and compiled with GCC 4.5.2 using the -O3 optimization flag. DataCutter uses an InfiniBand-aware MPI to leverage the high performance interconnect: here we used MVAPICH 1.1.

For testing purposes, we picked 4 large social network graphs from the SNAP dataset to perform the test at scale. The properties of the graphs are summarized in Table 4.3. For simulating the addition of the edges, we removed 50 edges from the graphs and added them back one by one. The streamed edges were selected randomly and uniformly. For comparability purposes, all the runs performed on the same graph use the same set of edges. The number of updates induced by that set of edges when applying filtering using identical vertices, biconnected component decomposition, and level filtering is given in Table 4.3. In the experiments, the data comes from a file,

Name	$\|V\|$	$\|E\|$	# updates	time(s)	speedup w.r.t [149] seq. non-incremental	speedup w.r.t. [149] seq. incremental
web-NotreDame	325,729	1,090,008	399,420	3.29	43,237	805
amazon0601	403,394	2,443,308	1,548,288	33.16	22,471	449
web-Google	916,428	4,321,958	2,527,088	71.20	45,860	578
soc-pokec	1,632,804	30,622,464	4,924,759	816.73	-	-

Table 4.3: Properties of the graphs we used in the experiments and execution time on a 64 node cluster.

and the Streaming Events are pushed to the system as quickly as possible so as to stress the system.

All the results presented in this section are extracted from a single run of STREAMER with proper parameters. The regularity in the plots indicates there would be a small variance on the runtimes, which induces a reasonable confidence in the significance of the quoted numbers. In the experiments, *StreamingMaster* and *Aggregator* run on the same node, apart from all the computational filters. Therefore, we report the number of worker nodes, but an extra node is always used.

To give an idea of the actual amount of computation, in the fourth column of Table 4.3, we report the time STREAMER spends to update the CC scores upon 50 edge insertions by using all 63 worker nodes. We also present the speedup of parallel implementation on 64 nodes with respect to sequential non-incremental computation and sequential incremental computation. The STREAMER framework is never sequential due to its distributed-memory nature and the pipelined parallelism, i.e., different

filters are always handled by different threads even in the most basic setting with no filter replication. (STREAMER uses at least the four filters of Figure 4.4, so at least four POSIX threads are always used.) Therefore, there is no sequential runtime for the STREAMER framework. When we mention the sequential time, it refers to our previous work [149], which runs sequentially using a single core of the same cluster. As all the execution times given in this section, the times in Table 4.3 do not contain the initialization time. That is the time measurement starts once STREAMER is idle, waiting to receive Streaming Events.

Basic performance results

Figure 4.10 shows the performance and scalability of the system in different configurations. The performance is expressed in number of updates per second. The framework obtains up to $11,000$ updates/sec on `amazon0601` and `web-Google`, $49,000$ updates/sec on `web-NotreDame`, and more than 750 updates/sec on the largest tested graph `soc-pokec`. It appears to scale linearly on the graphs `amazon0601` and `web-Google`, `soc-pokec`. For the first two graphs, it reaches a speedup of 456 and 497, respectively, with 63 nodes and 8 threads/node compared to the single node-single thread configuration. (The incremental centrality computation on `soc-pokec` with a single node was too long to run the experiment, but the system is clearly scaling well on this graph.) The last graph, `web-NotreDame`, does not exhibit a linear scaling and obtains a speedup of only 316.

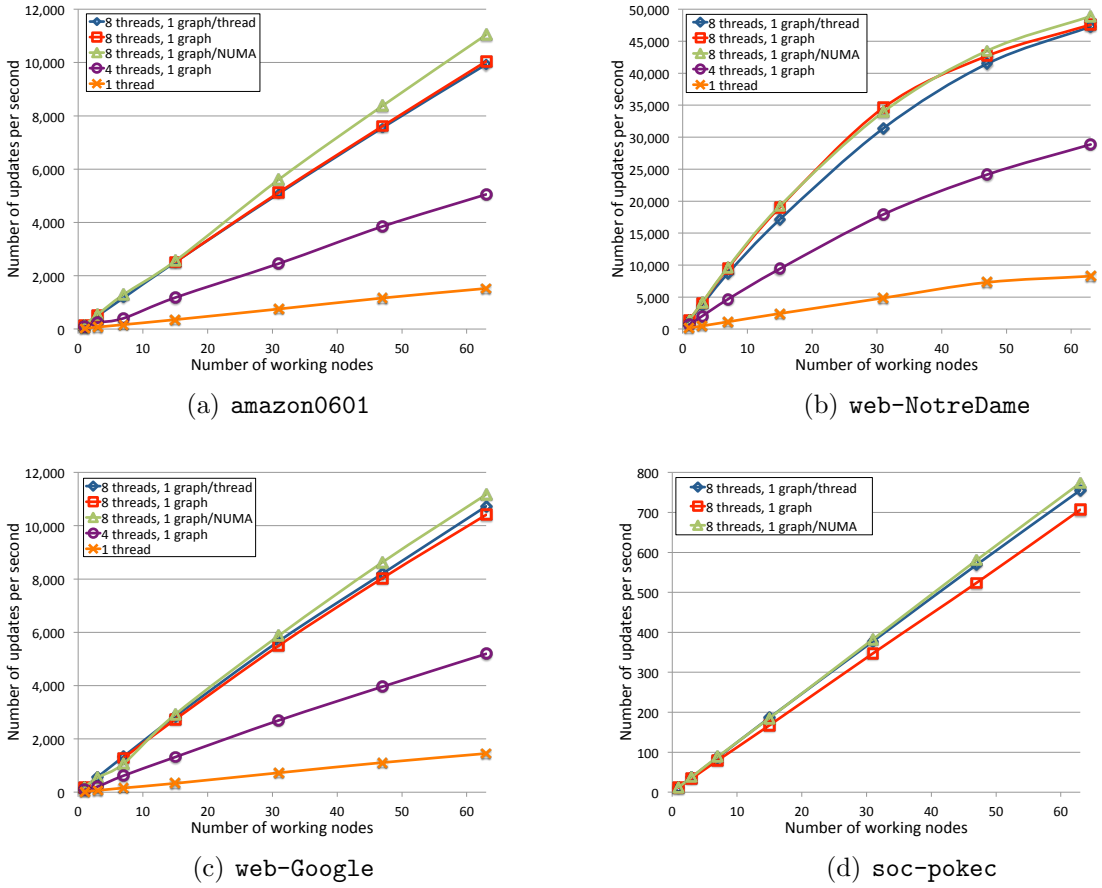
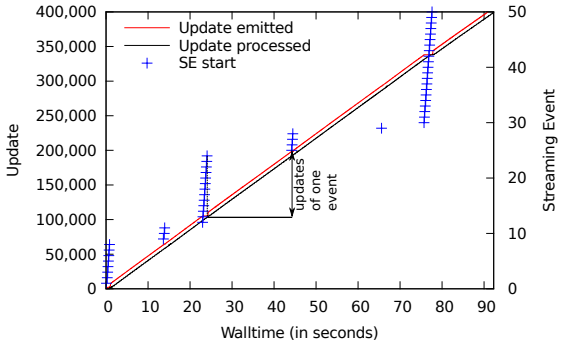


Figure 4.10: Scalability: the performance is expressed in the number of updates per second. Different worker-node configurations are shown. "8 threads, 1 graph/thread" means that 8 *ComputeCC* filters are used per node. "8 threads, 1 graph" means that 1 *Preparator* and 8 *Executor* filters are used per node. "8 threads, 1 graph/NUMA" means that 2 *Preparators* per node (one per NUMA domain) and 8 *Executors* are used.

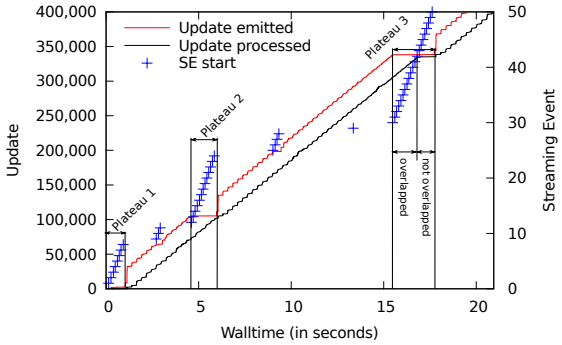
Table 4.4: The performance of STREAMER with 31 worker nodes and different node-level configurations normalized to 1 thread case (performance on `soc-pokec` is normalized to 8 threads, 1 graph/thread). The last column is the advantage of Shared Memory awareness (ratio of columns 5 and 3).

Name	4 threads	8 threads, 1 graph per			Shared Mem.
		thread	node	NUMA	awareness
web-NotreDame	3.69	6.46	7.13	6.99	1.08
amazon0601	3.26	6.75	6.81	7.45	1.10
web-Google	3.69	7.77	7.55	8.06	1.03
soc-pokec	-	1.00	0.92	1.01	1.01

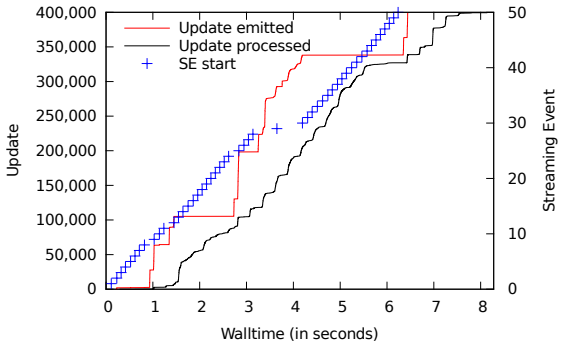
Let us first evaluate the performance obtained under different node-level configurations. Table 4.4 presents the relative performance of the system using 31 worker nodes while using 1, 4, or 8 threads per node. When compared with the single thread configuration, using 4 threads (the second column) is more than 3 times faster, while using 8 threads (columns 3–5) per node usually gives a speedup of 6.5 or more. Overall, having multiple cores is fairly well exploited. Properly taking the shared-memory aspect of the architecture into account (column 5) brings a performance improvement between 1% to 10% (the last column). In one instance (`web-Google` with a graph for each NUMA domain), we observed that the normalized performance is more than the number of cores. This can be explained by the fact that actually 10 threads are running on each computing node (8 *Executor* and 2 *Preparator*) which can lead to a higher parallelism.



(a) 3 worker nodes



(b) 15 worker nodes



(c) 63 worker nodes

Figure 4.11: Execution logs for `web-NotreDame` on different number of nodes. Each plot shows the total number of updates sent by *StreamingMaster* and processed by the *Executors*, respectively (the two lines), and the times at which *StreamingMaster* starts to process Streaming Events (the set of ticks).

Execution-log analysis

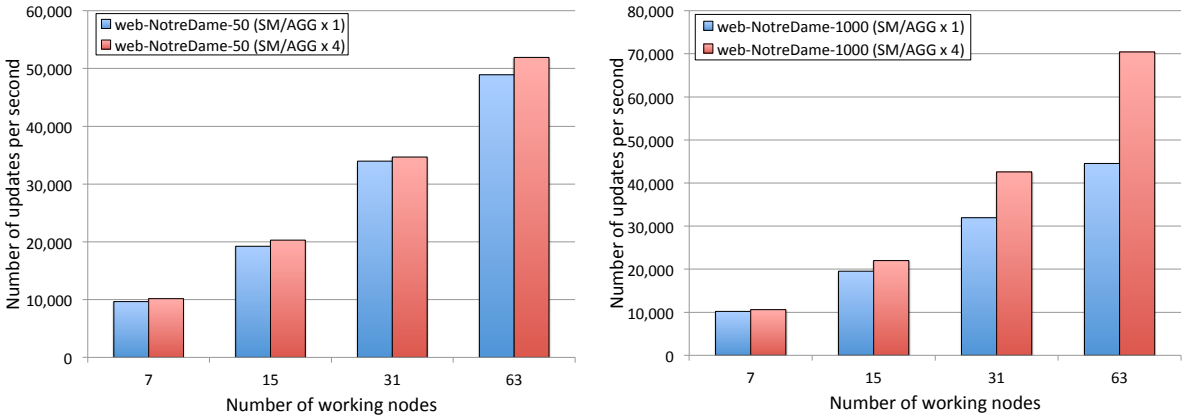
Here we discuss the impact of pipelined parallelism and the sub-linear speedup achieved on `web-NotreDame`. In Figure 4.11, we present the execution logs for that graph obtained while using 3, 15, and 63 worker nodes. Each log plot shows three data series: the times at which *StreamingMaster* starts to process the Streaming Events, the total number of updates sent by *StreamingMaster*, and the number of updates processed by the *Executors* collectively. The three different logs show what happens when the ratio of update produced and update consumed per second changes.

The first execution-log plot with 3 worker nodes (Fig. 4.11(a)) shows the amount of the updates emitted and processed as two perfectly parallel *almost straight* lines. This indicates that the runtime of the application is dominated by processing the updates. As the figure shows, the times at which the master starts processing the Streaming Events are not evenly distributed. As mentioned before, *StreamingMaster* starts filtering for the next Streaming Event as soon as it sends all the updates for the current one. In other words, the amount of updates emitted for a given Streaming Event can be read from the execution log as the difference of the y-coordinates of two consecutive "update emitted" points (the first line). In the first plot, we can see that 6 out of 50 Streaming Events (the ticks at the end of each partial tick-lines) incurred significantly much more updates than the others. While these events are being processed, the two lines stay straight and parallel, because in DataCutter,

writing to a downstream filter is a buffered operation. Once the buffer is full, the operation becomes blocking.

The second execution log with 15 worker nodes (Fig. 4.11(b)) shows a different picture. Here, the log is about 4 times shorter and the lines are not perfectly parallel. The number of updates emitted shows three plateaus for more than a second around times 0, 5, and 16 seconds. These plateaus exist because many consecutive Streaming Events do not generate a significant amount of updates; therefore, the master spends all its time by filtering the work for these Streaming Events.

The second plateau around time 5 seconds of the execution log with 15 worker nodes lasts 1.2 secs, and less than 100 updates are sent during that interval. However, as the plot shows, the worker nodes do not run out of work and process more than $25,000$ updates during the plateau. This is possible because the computation in STREAMER is pipelined. If the system were synchronous the worker nodes would spend most of that plateau waiting which yields a longer execution time and worse performance. In addition to the three large plateaus, cases with a few consecutive Streaming Events that lead to barely no updates are slightly visible around times 3 and 9. These two smaller cases are hidden by the pipelined parallelism. The third plateau is much longer than the second one (20 Streaming Events, 2.1 secs) and the worker nodes eventually run out of work halfway through the plateau. As can be seen in Fig. 4.10(b), the performance does not show linear scaling at 15 worker nodes; But it is still good, thanks to the pipelined parallelism.



(a) 50 edge insertions on web-NotreDame

(b) 1,000 edge insertions on web-NotreDame

Figure 4.12: Parallelizing *StreamingMaster* and *Aggregator*: the number of updates per second for `web-NotreDame` with 50 and 1,000 streaming events, respectively. The best node configuration from Figure 4.10, i.e., 8 threads, 1 graph/NUMA, is used for both cases.

When 63 worker nodes are used, the execution log (Fig. 4.11(c)) presents another picture. With the increase on the workers' processing power, *StreamingMaster* is now the main bottleneck of the computation. Two additional, considerably large plateaus appeared, and *StreamingMaster* starts to spend more than half of its time with the work filtering. However, during these times, the workers keep processing the updates, but at varied rates, due to temporary work starvation. The work filtering and the actual work are being processed mostly simultaneously showing that pipelined parallelism is very effective in this situation. Without the pipelined parallelism, the computation time would certainly be 2 secs longer, and 25% worse performance would be achieved.

We used the techniques described in Sections 4.4.2 and 4.4.3 (Figures 4.6 and 4.7) to replicate the *StreamingMaster* and *Aggregator* filters, respectively, and obtain a better performance when these filters becomes bottleneck throughout the incremental closeness centrality computation. The results on the `web-NotreDame` graph are given for 50 and 1,000 Streaming Events in Figure 4.12. As the figure shows, using four *StreamingMaster* and *Aggregator* filters instead of one yields around 6% improvement for 50 Streaming Events when 63 working nodes in the cluster are fully utilized. This small improvement is due to a lack of sufficient number of Streaming Events which generates a large amount of updates (see Figure 4.11). Hence, even with a large number of *StreamingMaster* and *Aggregator* filters, due to the load balancing problem on these filters, one cannot improve the performance more with 50 Streaming Events by just replicating them. Fortunately, in practice this number is usually much higher. In Figure 4.12(b), we repeated the same experiment for 1,000 Streaming Events. As the figure shows, the performance significantly increases when the filters are replicated. Furthermore, the percentage of the improvement increases when more nodes are used and reaches to 58% with 63 working nodes. This is expected since, with more cores for the *Executor* filters, the time spent for *StreamingMaster* and *Aggregator* becomes (relatively) more important. When applied on the other graphs, going from one *StreamingMaster* and *Aggregator* to four have not yield significant difference since these components were not bottlenecks. Therefore, we omitted those results here.

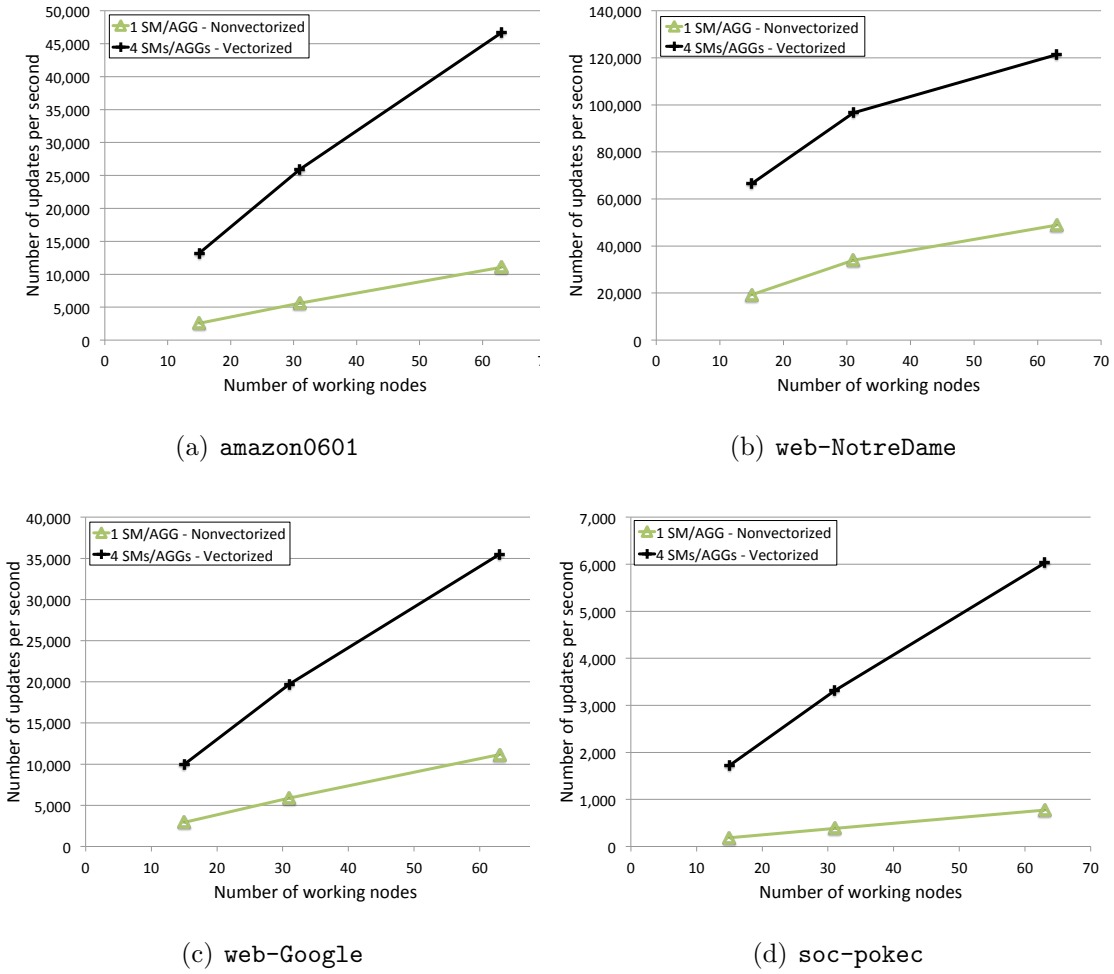
4.5.3 Plug-and-play filters: co-BFS

As stated above, thanks to filter-stream programming model, different filter implementations and various hardware such as GPUs can be used easily and efficiently if desired. Here, we show that using the SpMM-based approach described in Section 4.2.4, one can modify the *ComputeCC* filter in Figure 4.4 (or the *Executor* filters in Figure 4.5) to increase the performance. For this experiment, we swapped the *Executor* filter with one that uses the co-BFS algorithm which computes 32 BFSs from different sources concurrently. The results of the experiments with 15, 31, and 63 working nodes are shown in Figure 4.13. Using co-BFS (and coupled with multiple *StreamingMaster* and *Aggregator*) improves the performance of the regular version by a factor ranging from 2.2 to 9.3 depending on the graph and number of working nodes.

4.5.4 Illustrative example for closeness centrality evolution

In this section, we present a real world example to show how the closeness centrality scores of four researchers change over time in the temporal coauthor network obtained from DBLP[4]. We selected the four authors of this manuscript which have different experiences and looked at their closeness centrality score evolution from December 2009 to August 2014. We report the closeness centrality scores at the end of every 3 months by our incremental algorithms.

[4] http://dblp.uni-trier.de/xml/



(a) `amazon0601` (b) `web-NotreDame`

(c) `web-Google` (d) `soc-pokec`

Figure 4.13: co-BFS: the performance is expressed in the number of updates per second. The best worker-node configuration, "8 threads, 1 graph/NUMA", is used for the experiments.

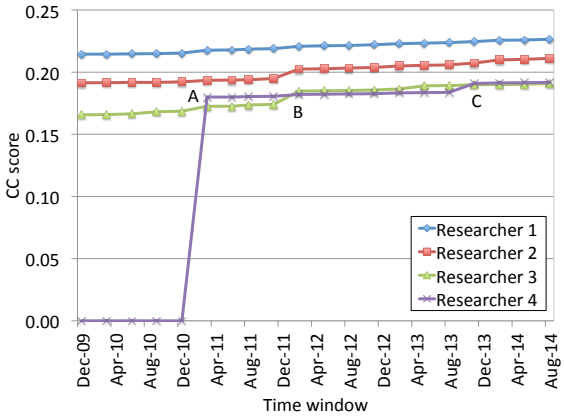


Figure 4.14: Closeness centrality score evolution in DBLP coauthor network

Figure 4.14 shows how the CC score changes when time passes. Researcher 4 is a PhD student who started in September 2010 and his first paper was published at the beginning of 2011. Point A shows the impact of the first paper on his CC score. Researcher 3 joined the team as a postdoc in September 2011. His first paper with the team members was published in early 2012 (Point B). We can observe that this publication increased his CC score, making him more central in the DBLP coauthor network. This publication also effected the centrality score of Researcher 2, who was another postdoc of the team at the time. Another significant point in the figure is point C, which corresponds to the publication of Researcher 4 as a result of his internship in an different institute. This publication made Researcher 4 more central since he is connected to new researchers in the DBLP coauthor network. Apart from

those important milestones, we can see that there is a steady increase in CC scores of the four researchers.

Summary of the experimental results

The experiments we conducted shows that STREAMER can scale up and efficiently utilize our entire experimental cluster. By taking the hierarchical composition of the architecture into account (64 nodes, 2 processors per node, 4 cores per processor) and not considering it as a regular distributed machine (a 512-processor MPI cluster), we enabled processing of larger graphs and obtained 10% additional improvement. Furthermore, the pipelined parallelism proved to be extremely necessary while using a large amount of nodes in a concurrent fashion.

Replicating the *ComputeCC* filter leads to significant speedup. Yet, the bottleneck eventually becomes the filters that cannot be replicated automatically. For filters where the ordering of the messages is important, we can substitute an alternative filter architecture to alleviate the bottleneck and make the whole analysis pipeline highly parallel.

The flexibility of the filter-stream programming model allows to easily substitute a component of the application by an alternative implementation. For instance, one can use modern vectorization techniques to improve the performance by a significant factor. Similarly, one could have an alternative implementation which use different type of hardware such as accelerators.

For the three of the graphs `web-NotreDame`, `amazon0601` and `web-Google`, a reference sequential time is known from [149] for both the non-incremental and the incremental cases. STREAMER using 63 worker nodes (8 cores per node), 4 *StreamingMaster* and 4 *Aggregators* and co-BFS computing filters improved the runtime of the incremental algorithm by a factor of 805, 449 and 578 respectively on the three graphs. Compared to a sequential non-incremental computation of the closeness centrality value, STREAMER improves the runtime by a factor ranging from 22471 to 45860. These numbers are reported in Table 4.3.

4.6 Related Work

To the best of our knowledge, there are only two works on maintaining centrality in dynamic networks. Yet, both are interested in betweenness centrality. Lee et al. proposed the QUBE framework which uses a BCD and updates the betweenness centrality values in case of edge insertions and deletions in the network [104]. Unfortunately, the performance of QUBE is only reported on small graphs (less than 100K edges) with very low edge density. In other words, it only performs significantly well on small graphs with a tree-like structure having many small biconnected components.

Green et al. proposed a technique to update the betweenness centrality scores rather than recomputing them from scratch upon edge insertions (can be extended to edge deletions) [76]. The idea is to store the whole data structure used by the previous computation. However, as the authors stated, it takes $\mathcal{O}(n^2 + nm)$ space to

store all the required values. Compared to their work, our algorithms are much more practical since the memory footprint of linear.

4.7 Summary

In this chapter, we propose the first algorithms to achieve fast updates of exact closeness centrality values on incremental network modification at such a large scale. Our techniques exploit the spike-shaped shortest-distance distributions of these networks, their biconnected component decomposition, and the existence of nodes with identical neighborhood. In large networks with more than $500K$ edges, the proposed techniques bring 99 times speedup on average. For the temporal DBLP coauthorship graph, which has the most edges, we reduced the centrality update time from 1.3 days to 4.2 minutes.

Furthermore, we parallelized our algorithms and proposed Streamer, a distributed memory framework which guarantees the correctness of the CC scores, exploits replicated and pipelined parallelism, and takes the hierarchical architecture of modern clusters into account. Using Streamer on a 64 nodes, 8 cores/node cluster, we reached a speedup of 497. Furthermore, the system is fully scalable as each of its components can be made to use an arbitrary number of nodes. Also, we showed that we can easily use alternative implementation of the BFS computations to allow the use of novel algorithmic techniques or hardware. Using Streamer on a 64 nodes, 8 cores/node cluster, we reached almost linear speedup in the experiments and the performance are orders of magnitude higher than the non-incremental computation.

Maintaining the closeness centrality of large and complex graph in real-time is now within our grasp.

Chapter 5: Streaming k-core Decomposition

The k-core decomposition of a graph maintains, for each vertex, the max-k value: the maximum k value for which a k-core containing the vertex exists. This decomposition enables one to quickly find the k-core containing a given vertex for a given k. Algorithms for creating k-core decomposition of a graph in time linear to the number of edges in the graph exist [22]. For applications that manage dynamic graphs, applying such algorithms for every edge insertion and removal is prohibitive in terms of performance. Furthermore, batch processing takes away the ability to react to changes quickly – one of the key benefits of stream processing [172].

5.1 Introduction

In this chapter, we develop streaming algorithms for k-core decomposition of graphs. In particular, we develop algorithms to update the decomposition as edges are inserted into and removed from the graph (vertex additions and removals are trivial extensions). There are a number of challenges in achieving this. The first is a theoretical one: determining a small subset of vertices that are guaranteed to contain all vertices that may have their max-k values changed as a result of an insertion or

removal. The second is a practical one: finding algorithms that can efficiently update the max-k values using this subset. Last but not the least, we have to understand the impact of the graph structure on the performance of such streaming algorithms.

We address these challenges by developing the first incremental k-core decomposition algorithm for streaming graph data, where we efficiently process a small subgraph for each change. We develop a number of variations of our algorithm and empirically show that incremental processing provides a significant reduction in run-time compared to non-incremental alternatives, reaching 6 orders of magnitude speedup for a graph of size of around 16 million. We showcase the efficiency of our algorithms on different types of real and synthetic graphs at different scales and study the impact of graph structure on the performance of algorithm variations.

In summary, we make the following major contributions:

- We identify a small subset of vertices that have to be visited in order to update the max-k values in the presence of edge insertions and deletions.
- We develop various algorithms to update the k-core decomposition incrementally. To the best of our knowledge, these are the first such incremental algorithms.
- We present a comparative experimental study that evaluates the performance of our algorithms on real-world and synthetic data sets.

The rest of this chapter is organized as follows. Section 5.2 gives the background on k-core decomposition of graphs. Section 5.3 introduces our theoretical findings that

facilitate incremental k-core decomposition. Section 5.4 introduces several new algorithms for incremental maintenance of a graph's k-core decomposition. Section 5.5 provides discussions on implementation details. Section 5.6 gives a detailed experimental evaluation of our algorithms. Section 5.7 reports related work, and Section 5.8 concludes the chapter.

5.2 Background

In this work, we focus on incremental maintenance of k-core decomposition of large networks modeled as undirected and unweighted graphs. Here, we start by giving several definitions that are used throughout the chapter as part of our theorems and proofs.

Let G be an undirected and unweighted graph. For a vertex-induced subgraph, which consists of some of the vertices of the original graph and all of the edges that connect them in the original, $H \subseteq G$, $\delta[H]$ denotes the minimum degree of H, defined as the minimum number of neighbors a vertex in H has (i.e., $\delta[H] = min\{\delta_H(u) : u \in H\}$, where $\delta_H(u)$ denotes the number of neighbors of a vertex u in H). As a result, any vertex in H is adjacent to at least $\delta[H]$ other vertices in H and there is no other value larger than $\delta[H]$ that satisfies this property.

Definition 3. *If H is a connected graph with $\delta(H) \geq k$, we say that H is a* seed *k-core of G. Additionally, if H is* ***maximal*** *(i.e., $\nexists H'$* s.t. *$H \subset H' \wedge H'$ is a seed k-core of G), then we say that H is a k-core of G.*

Observation 1. *Let H be a k-core that contains the vertex u. Then, H is unique in the sense that there can be no other k-core that contains u.*

We denote the unique k-core that contains u as H_k^u.

Definition 4. *The* maximum *k-core associated with a vertex u, denoted by H^u, is the k-core that contains u and has the largest $k = \delta[H^u]$ (i.e., $\nexists H$ s.t. $u \in H \wedge H$ is an l-core $\wedge\ l > k$). The* maximum *k-core number of u (also called the K value of u), denoted by $K(u)$, is defined as $K(u) = \delta[H^u]$.*

Observation 2. *If H is a k-core in graph G, then there exists one and only one $(k-1)$-core $H' \supseteq H$ in G.*

Observation 3. *A vertex u with $K(u) = k$ takes part in cores $H_k^u \subseteq H_{k-1}^u \subseteq H_{k-2}^u, \ldots, \subseteq H_1^u$ by Observation 2.*

Building the core decomposition of a graph G is basically the same problem as finding the set of *maximum* k-cores of all vertices in G. The following corollary shows that given the K values of all vertices, k-core of any vertex can be found for any k.

Algorithm 11: FINDKCOREDECOMPOSITION($G(V, E)$)

```
   Data: G: the graph
1  Compute δ_G(v) (i.e., the degree) for all vertices v ∈ V
2  Order the set of vertices v ∈ V in increasing order of δ_G(v)
3  for each v ∈ V do
4  |  K(v) ← δ_G(v)
5  |  for each (v, w) ∈ E do
6  |  |  if δ_G(w) > δ_G(v) then
7  |  |  |  δ_G(w) ← δ_G(w) − 1
8  |  Reorder the rest of V accordingly
9  return K
```

Corollary 5. *Given $K(v)$ for all vertices $v \in G$ and assuming $K(u) \geq k$, the k-core of a vertex u, denoted by H_k^u, consists of u as well as any vertex w that has $K(w) \geq k$ and is reachable from u via a path P such that $\forall_{v \in P}$, $K(v) \geq k$. H_k^u can be found by traversing G starting at u and including each traversed vertex w to H_k^u if $K(w) \geq k$.*

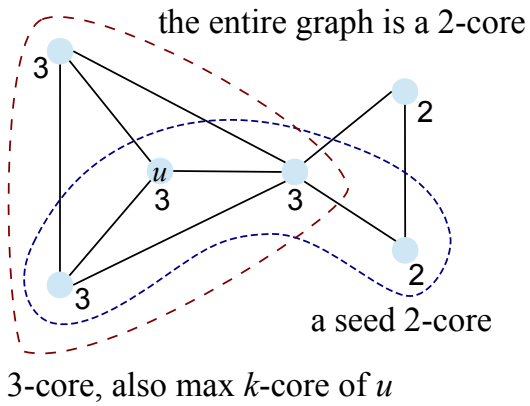


Figure 5.1: Illustration of k-core concepts.

Intuitively, in Corollary 5, all the traversed vertices are in H_k^u due to maximality property of k-cores, and all the vertices in H_k^u are traversed due to the connectivity property of k-cores, both based on Definition 3.

Thus, the problem of maintaining the k-core decomposition of a graph is equivalent to the problem of maintaining its K values, by Corollary 5. The algorithm for constructing the k-core decomposition of a graph from scratch is based on the following property [161]: To find the k-cores of a graph, all vertices of degree less than k and their adjacent edges are recursively deleted. We provide its pseudo-code in Algorithm 11 for completeness.

Figure 5.1 illustrates concepts related to k-core decomposition. In the sample graph, we see the K values of the vertices printed next to them, which is simply the k-core decomposition of the graph. We see a vertex labeled u. A seed 2-core that

contains u is also shown. Moreover, the entire graph is the 2-core of u, i.e., $G = H_2^u$. The figure further shows a 3-core of u, that is H_3^u, which happens to be its max-k core, that is $H_3^u = H^u$. Note that $H_3^u \subseteq H_2^u$.

5.3 Theoretical Findings

In this section, we introduce our theoretical findings. These results facilitate incremental maintenance of the k-core decomposition of a graph. Since our incremental algorithms rely on finding a subgraph and processing it, we prove a number of theorems that can be used to find a small subgraph that is guaranteed to contain all the vertices whose K values change after an update.

Observation 4. *Let $G = (V, E)$ be a graph and $u, v \in V$. If there is an edge $e \in E$ between u and v and if $K(u) > K(v)$, then $e \notin H^u$ and $e \in H^v$, by Corollary 5.*

Theorem 4. *If an edge is inserted to or removed from graph $G = (V, E)$, then the K value of vertex $u \in V$ can change by at most 1.*

Proof. We first prove the insertion case. Assume that after the insertion of edge e, $K(u) = m$ is increased by n to $K_+(u) = m + n$, where $n > 1$. Let us denote the max k-core of u after the insertion as H_+^u, and before insertion as H^u. It must be true that $e \in H_+^u$, as otherwise H_+^u forms a seed $m + n$-core before the insertion as well, which is a contradiction. Let $Z = H_+^u \setminus e$. If Z is not disconnected, then it must form an $m + n - 1$-core, since the degree of its vertices can decrease by at most 1 due to removal of a single edge. This leads to a contradiction since $m + n - 1 > m$ and H^u is maximal. In the disconnection case, each one of the resulting two connected components must be a seed $m + n - 1$-core as well, since the degree of a vertex can reduce by at most one in each component. Furthermore, since e is the only edge between the two disconnected components, the vertices must still have at least $m + n - 1$ neighbors in their respective components. One of these components must contain u, which is again contradiction

Next, we prove the removal case. Assume $K(u)$ is decreased by n after edge e is removed, where $n > 1$. Adding e back to the graph increases the K value of u by n, which is not possible, as shown in the first part of the proof (i.e., a contradiction). □

Theorem 5. *If an edge (u, v) is inserted to or removed from $G = (V, E)$, where $u, v \in V$ and $K(u) < K(v)$, then $K(v)$ cannot change.*

Proof. We first prove the insertion case. Assume that $K(v) = n$ increases and so becomes $K_+(v) = n + 1$ by Theorem 4. Then we have $e \in H^v_+$ and consequently $u \in H^v_+$. However, $K(u) < n$ before insertion and $K_+(u)$ can be at most n after insertion (Theorem 4), implying that u cannot be in a seed $n + 1$-core, i.e., a contradiction.

For the removal case, assume that $K(v) = n$ decreases and becomes $K_-(v) = n-1$ by Theorem 4. Inserting (u, v) back to the graph should increase the K value of v to $K(v) = n$. We must also have $e \in H^v$ and thus $u \in H^v$. But this is a contradiction due to Observation 4, since $K(u) < K(v)$ and $u \notin H^v$. □

From Theorem 5, we can say that when an edge (u, v) is inserted into or removed from the graph, $K(u)$ can change by at most 1 if $K(u) \leq K(v)$, or stay the same otherwise.

Theorem 6. *If an edge (u, v) is inserted into $G = (V, E)$, where $u, v \in V$, then all of the vertices whose K values have changed should form a connected subgraph $G' \subset G \cup (u, v)$. Similarly, if an edge (u, v) is removed from $G = (V, E)$, where $u, v \in V$, then all the vertices whose K values have changed should form a connected subgraph $G'' \subset G$.*

Proof. We prove the insertion case first. Assume that the updated vertices do not form a connected subgraph. Then, there are at least 2 non-overlapping subgraphs of updated vertices, S_1 and S_2. Since there is only one edge insertion, only one of these subgraphs, say S_1, can have a vertex who gets a new neighbor in G. Then S_2 does not have any vertex that has its degree changed. This is a contradiction, because if a vertex has its K value increased, then it must have either gained a new neighbor (increased degree) or at least one of its existing neighbors must have its K value increased. Applying this recursively, we must reach a vertex whose K value is increased due to gaining a new neighbor. However, for S_2, there is no such vertex since only reachable vertices whose K values have increased are in S_2 and none of them have their degrees changed.

For the removal case, assume that the updated vertices do not form a connected subgraph. Then, there are at least 2 non-overlapping subgraphs of updated vertices, S_1 and S_2. Since there is only one edge removal, only one of these subgraphs, say S_1, can have a vertex who loses a neighbor in G. Then S_2 does not have any vertex

that has its degree changed. This is a contradiction, because if a vertex has its K value decreased, then it must have either lost a neighbor (decreased degree) or at least one of its existing neighbors must have its K value decreased. Applying this recursively, we must reach a vertex whose K value is decreased due to losing an existing neighbor. However, for $S2$, there is no such vertex since only vertices that can be reached and whose K value has decreased are in $S2$ and none of them have their degrees changed. □

Theorem 7. *Given a graph $G = (V, E)$, if an edge (u, v) is inserted (removed) and $K(u) \leq K(v)$, then only the vertices $w \in V$, that have $K(w) = K(u)$ and are reachable from u via a path that consists of vertices with K values equal to $K(u)$,* ***may*** *have their K values incremented (decremented).*

Proof. Before looking at the insertion and removal, we note that if the K value of any vertex in G increases (decreases) due to the insertion (removal) of (u, v), then $K(u)$ must have increased (decreased) as well. This follows from the recursive argument in Theorem 6, as otherwise none of the vertices that have their K values changed will have their degree changed.

For the insertion case, we first prove that for a vertex $w \in V$ such that $K(w) \neq K(u)$, $K(w) = m$ cannot change. We consider two cases: (i) where $K(w) < K(u)$ and (ii) where $K(w) > K(u)$.

For the $K(w) > K(u)$ case, assume $K(w)$ increases ($K_+(w) = m + 1$). We must have $(u, v) \in H^w_+$, as otherwise H^w would not be a max m-core before insertion. However, this is not possible since $K_+(w) > K_+(u)$, i.e., a contradiction.

For the $K(w) < K(u)$ case, assume $K(w)$ increases ($K_+(w) = m + 1$). Then we have $(u, v) \in H^w_+$, as otherwise H^w would not be a max m-core before insertion. We know that $m + 1 \leq K(u) \leq K(v)$, which implies $K_+(w) < K_+(u) \leq K_+(v)$. Removing (u, v) from H^w_+ decreases the degrees of u and v by one, which can reduce their K value to at least $m + 1$. This means $H^w_+ \setminus (u, v)$ is a seed $m + 1$-core before the insertion, which is a contradiction.

We proved that only vertices with $K(w) = K(u)$, say $L \subseteq V$, **can** have their K values incremented. Furthermore, we know that all those vertices form a connected subgraph (Theorem 6). Since we have $u \in L$ as well, the insertion proof is complete.

We use similar arguments for the removal case. Again, we consider two cases.

For the $K(w) < K(u)$ case, assume $K(w)$ decreases ($K_-(w) = m - 1$). Say that we insert (u, v) back into the graph. The K value of w cannot increase in this case since $K_-(w) < K_-(u)$, and this is a contradiction, as shown in insertion part above.

For the $K(w) > K(u)$ case, assume $K(w)$ decreases ($K_{-}(w) = m - 1$). We know that $(u, v) \notin H^w$ since $u \notin H^w$ due to $K(u) < K(w)$. Thus, H^w is still an m-core after the removal, creating a contradiction.

We proved that only the vertices that have $K(w) = K(u)$, say $L \subseteq V$, **may** have their K values decremented. Furthermore, by Theorem 6, we know that all those vertices form a connected subgraph. Since we have $u \in L$, the removal proof is complete. □

Summary In this Section, we showed that if an edge (u, v) is inserted into/removed from a graph, then the K value of u can change only if $K(u) \leq K(v)$. Let us call u the root. In case $K(u) = K(v)$, then either u or v is taken as the root. In addition, we showed that any vertex that may have its K value updated must have a K value that is equal to that of the root, and must be connected to the root via a path that contains only the vertices that have the same K value. We rely on these results in the next section.

5.4 Incremental Algorithms

In this section, we introduce four algorithms to incrementally maintain the K values of vertices when a single edge is inserted or removed. The subcore (Section 5.4.1) and purecore (Section 5.4.2) algorithms are basic applications of the theoretical results given in the previous section, are easy to implement, and form a baseline for evaluating the performance of the traversal algorithm (Section 5.4.3). The traversal algorithm relies on additional ideas that aggressively cut the search space, but is more involved than the earlier two. For edge insertion case, we also introduce the *generic multihop*

traversal algorithm (Section 5.4.4) which generalizes the traversal algorithm to utilize multihop information residing on vertices.

5.4.1 The Subcore Algorithm

Our first algorithm for maintaining the K values of vertices when a single edge is inserted or removed is based on Theorem 7. We define a new subgraph as follows:

Definition 5. *Given a graph $G = (V, E)$ and a vertex $u \in V$, the* subcore *of u, also denoted as S_u, is a set of vertices $w \in V$ that have $K(w) = K(u)$ and are reachable from u via a path that consists of vertices with their K values equal to $K(u)$.*

Given a graph $G = (V, E)$ and the K values of all $w \in V$, if an edge (u_1, u_2) is inserted to E, Algorithm 13 updates the K values. Similarly, if an edge (u_1, u_2) is removed from E, Algorithm 14 updates the K values. Both algorithms make use of Definition 5.

The basic idea is to locate the subcore of the root vertex and apply a process very similar to Algorithm 11 on the subcore. Algorithm 12 provides the pseudo code for finding the subcore. To find the subcore, we perform a BFS traversal and collect all vertices reachable from the root through vertices having the same K value as the root. During this process, we also collect the *core degree* (`cd`) values for each vertex in the subcore. The core degree of a vertex is its degree in its max-core and determines if a vertex can change its K value or not. As a result, the `cd` of a vertex simply counts the number of its neighbors with a K value equal to or greater than the K value of the root. Core degrees help us eliminate vertices that cannot be part of a $k + 1$ core, where k is the core value of the root. In particular, if the core degree is not

Algorithm 12: FINDSUBCORE$(G(V,E), K(), u)$

```
   Data: G: the graph, K: max-k values, u: the vertex
 1 H(V', E') ← empty graph; Q ← empty queue
 2 cd[v] = 0; visited[v] = false, ∀v ∈ V                    ▷ Lazy init
 3 k ← K(u)                                  ▷ Remember K value of the root
 4 Q.push(u); visited[u] ← true
 5 while not Q.empty() do
 6 |   v ← Q.pop(); V'.push(v)
 7 |   for each (v, w) ∈ E do
 8 |   |   if K(w) ≥ k then
 9 |   |   |   cd[v] ← cd[v] + 1
10 |   |   |   if K(w) = k and not visited[w] then
11 |   |   |   |   Q.push(w); E'.push((v, w))
12 |   |   |   |   visited[w] ← true
13 return H and cd
```

larger than k, we can eliminate the vertex from consideration. Once it is eliminated, it results in decrementing the core degree values of its neighbors in the subcore and the process can be repeated. Similar to Algorithm 11, this has to be performed in increasing order of the core degree values.

Algorithm 13 shows how the subcore and the `cd` values are used to update the K values on an edge insertion. We order the `cd` values of the vertices in the subcore in increasing order. At each step, we pick the unprocessed vertex with the smallest `cd` value from the subcore. If it has a `cd` value less than or equal to the root's K value, say k, then it cannot be part of a $k+1$-core. Thus, for each of its neighbors in the subcore that have a higher `cd`, we decrement the neighbor's `cd` by 1, since the vertex being processed cannot be part of a higher core. We reorder the remaining vertices

Algorithm 13: SUBCORE: INSERTEDGE$(G(V,E), K(), u_1, u_2)$

```
   Data: G: the graph, K: max-k values, (u1, u2): inserted edge
 1 r ← u1                                                    ▷ Set the root
 2 if K(u2) < K(u1) then r ← u2
 3
 4 G ← G ∪ (u1, u2)                                  ▷ Add the edge into G
 5 H, cd ← FINDSUBCORE(G, K, r)                          ▷ Find subcore
   ▷ Now update the K values of the vertices in H
 6 k ← K(r)                                  ▷ Remember K value of the root
 7 Sort cd values in increasing order (using bucket sort)
 8 for each v ∈ H in order do
 9     if cd[v] ≤ k then                          ▷ Cannot be part of a k+1-core
10         for each (v, w) ∈ H do
11             if cd[w] > cd[v] then
12                 cd[w] ← cd[w] − 1
13                 Reorder cd values accordingly
14     else                    ▷ All remaining vertices become part of k+1-core
15         for each w ∈ H do
16             K(w) ← k + 1
17         break
```

based on their updated `cd` values. Otherwise, that is if the current vertex has a `cd` value larger than k, all remaining vertices must also have their `cd` values larger than k, which means we can form a seed $k+1$ core with them. We increment their K values, completing the insertion.

Algorithm 14 shows how the subcore and the `cd` values are used to update the K values in the case of a removal. Unlike Algorithm 13, here we need to perform two subcore searches when the K values of the vertices incident upon the removed edge are the same, since the removal separates them. Once we locate the subcore, the process is very similar to that of the insertion. We pick the unprocessed vertex

with the smallest `cd` value from the subcore and if it has a `cd` value less than the K value of the root, say k, then it cannot be part of a k-core anymore. As a result, we decrement its K value and for each of its neighbors in the subcore that have a higher `cd`, we decrement the neighbor's `cd` by one, since the vertex currently being processed cannot be part of a higher core. After this, we reorder the remaining vertices based on their `cd` values. Otherwise, if the current vertex has a `cd` value larger than or equal to k, then all remaining vertices must also have their `cd` values larger than or equal to k, which means that we can still form a seed k core with them. Thus, we stop processing and complete the removal.

5.4.2 The Purecore Algorithm

In Section 5.4.1, the subcore algorithm relied only on the K values of the vertices to locate a small subgraph that contains all the vertices that can have their K values changed. In this section, we look at the *purecore* algorithm that takes advantage of additional information about each vertex, so that a smaller set of candidate vertices can be located, reducing the overall cost of the algorithm. For this purpose, we define the *maximum-core degree* of a vertex.

Definition 6. *The maximum-core degree of a vertex u, denoted as $MCD(u)$, is defined as the number of u's neighbors, w, such that $K(u) \leq K(w)$.*

The maximum-core degree of a vertex differs from the core degree of a vertex by the fact that it is not defined in terms of the root vertex of an insertion. If the MCD value of a vertex is not greater than its K value, and no new adjacent edge is inserted,

Algorithm 14: SUBCORE:REMOVEEDGE$(G(V,E), K(), u_1, u_2)$

Data: G: the graph, K: max-k values, (u_1, u_2): inserted edge

$r \leftarrow u_1$ ▷ Set the root
if $K(u_2) < K(u_1)$ **then** $r \leftarrow u_2$

$G \leftarrow G \setminus (u_1, u_2)$ ▷ Remove the edge from G
if $K(u_1) \neq K(u_2)$ **then**
    $H, \texttt{cd} \leftarrow$ FINDSUBCORE(G, K, r) ▷ Find subcore
else
    $H_1, \texttt{cd}_1 \leftarrow$ FINDSUBCORE(G, K, u_1) ▷ Find subcore of u_1
    $H_2, \texttt{cd}_2 \leftarrow$ FINDSUBCORE(G, K, u_2) ▷ Find subcore of u_2
    $H \leftarrow H_1 \cup H_2$; $\texttt{cd} \leftarrow \texttt{cd}_1 \cup \texttt{cd}_2$
▷ Now update the K values of the vertices in H
$k \leftarrow K(r)$ ▷ Remember K value of the root
Sort cd values in increasing order (using bucket sort)
for each $v \in H$ *in order* **do**
    if $\texttt{cd}[v] < k$ **then** ▷ Cannot be part of a k-core anymore
        $K(v) \leftarrow k - 1$
        for each $(w, v) \in H$ **do**
            if $\texttt{cd}[w] > \texttt{cd}[v]$ **then**
                $\texttt{cd}[w] \leftarrow \texttt{cd}[w] - 1$
                Reorder cd values accordingly
    else break ▷ All remaining vertices still in a k-core

then there is no way for this vertex to increment its K value, because the number of neighbor vertices in a higher core will not be enough. Therefore, it is used to test whether a vertex can increment its K value or not, upon a new edge insertion.

Observation 5. *For a given graph $G = (V, E)$ and a vertex $u \in V$, $MCD(u) \geq K(u)$.*

The observation follows simply from the definition of k-core, since $MCD(u) < K(u)$ would mean u cannot participate in a k-core with $K(u) = k$, leading to a contradiction. Note that $MCD(u)$ is simply an upper bound on $K(u)$.

We reduce the *subcore*, described in Definition 5, to a *purecore* by putting an extra condition regarding MCD values. The basic idea is that, if a vertex in the subcore does not have a MCD value greater than the K value of the root, it means that the vertex does not have enough neighbors that can participate in a higher core.

Definition 7. *Given a graph $G = (V, E)$ and a vertex $u \in V$, the* purecore *of u, denoted as P_u, is the set of vertices $w \in V$ that have $K(w) = K(u)$ and $MCD(w) > K(u)$, and are reachable from u via a path that consists of vertices with K values equal to $K(u)$ and MCD values greater than $K(u)$.*

Algorithm 15 finds the purecore P_u of a vertex u.

Theorem 8. *Given a graph $G = (V, E)$, if an edge (u, v) is inserted and $K(u) \leq K(v)$, then only the vertices $w \in P_u$* ***may*** *have their K values incremented.*

Proof. When an edge (u, v) is inserted to the graph and $K(u) \leq K(v)$, then the K value of a vertex $w \in S_u$, where $w \neq u$, cannot increment if $MCD(w) = K(w)$. Assume $K(w)$ increments, then $MCD(w)$ has to increment as well, and for this to happen either w should get a new neighbor, which is not possible since $w \neq u$, or some of its neighbors should have their K values decreased, which is not possible as no edges were removed. □

Algorithm 15: findPurecore$(G(V, E), K(), u)$

```
   Data: G: the graph, K: max-k values, u: the vertex
 1 H(V', E') ← empty graph; Q ← empty queue
 2 cd[v] = 0; visited[v] = false, ∀v ∈ V                          ▷ Lazy init
 3 k ← K(u)                                          ▷ Remember K value of the root
 4 Q.push(u); visited[u] ← true
 5 while not Q.empty() do
 6 |   v ← Q.pop(); V'.push(v)
 7 |   for each (v, w) ∈ E do
 8 |   |   if K(w) > k or (K(w) = k and
 9 |   |                                          MCDegree(G, K, w)> k) then
10 |   |   |   cd[v] ← cd[v] + 1
11 |   |   |   if K(w) = k and not visited[w] then
12 |   |   |   |   Q.push(w); E'.push((v, w));
13 |   |   |   |   visited[w] ← true
14 return H and cd
```

With purecore, the algorithm to update the K values of vertices, when edge (u, v) is inserted, is the same as Algorithm 13, except that Algorithm 15 (findPurecore) is used in place of Algorithm 12 (findSubcore).

When an edge (u, v) is removed from the graph and $K(u) \leq K(v)$, then the K value of any vertex $w \in S_u$ can potentially decrement. Note that $MCD(w)$ can decrease if either w loses a neighbor, which is the case for u, or K value of some neighbor of w decrements, which is the case for neighbors of u when $K(u)$ decrements. As a result, for removal, we do not rely on the purecore.

5.4.3 The Traversal Algorithm

We now present the traversal algorithm that visits an even smaller subgraph to update the k-value decompostion. First, we introduce an optimization to speedup the computation of the MCD values and then an additional metric to further scope the search.

Residential Core Degrees

In Section 5.4.2, we find a smaller set of candidate vertices to be updated by using more information about each vertex. Using more information, such as the MCD values, requires more computation in Algorithm 15. Thus, for a vertex u, when the size of P_u is **large** and close to the size of S_u, Algorithm 15 turns out to be more expensive than Algorithm 12. To alleviate this problem, we make two types of core degree values to constantly reside in memory (i.e., *residential*). We maintain the MCD values, introduced in Definition 6, and the PCD values of vertices defined as follows.

Definition 8. *The purecore degree of a vertex u, denoted as $PCD(u)$, is the number of u's neighbors, w, such that* ***either*** $K(u) = K(w)$ *and* $MCD(w) > K(u)$ ***or*** $K(u) < K(w)$.

For a vertex v, its purecore degree $PCD(v)$ is the number of neighbors w it has that either has a higher K value than v or has the same K value but in turn has enough neighbors to potentially increase its K value (in case an insertion was made and the K values are to be updated). The PCD value of a vertex represents its potential number of neighbors in a next max-core. It is a stronger indicator than its

MCD value for showing eligibility to increase the K value and also useful, because if $PCD(v) \leq k$ where k is the K value of the root, then v cannot increment its K value.

Maintaining the MCD and PCD values of vertices after each insertion and removal should be done efficiently so that unnecessary updates of those values are avoided. In general, the MCD value of a vertex is based on the K values of its neighbors, as seen from Definition 6, and the PCD value of a vertex is based on the K and MCD values of its neighbors, as described in Definition 6. Observation 6 gives a rule of thumb for MCD and PCD maintenance.

Observation 6. *For a graph $G = (V, E)$, when the K value of a vertex $u \in V$ changes, the MCD values of vertices u, v can change, where $(u, v) \in E$. When the MCD value of a vertex $u \in V$ changes, the PCD values of vertices v can change, where $(u, v) \in E$. As a result, when the K value of a vertex $u \in V$ changes, the PCD values of vertices u, v, w can change, where $(u, v), (v, w) \in E$.*

In summary, the observation says that a K value update can result in changes in the MCD values within the 1-hop neighborhood of the vertex, whereas changes in the PCD values can happen within the 2-hop neighborhood.

Based on Observation 6, when an edge (u, v) is inserted into or removed from a graph $G = (V, E)$, we first recompute the MCD value of the root vertex u and the PCD values of its neighbors. Next, we apply the algorithm to update the K values of vertices. Last, we do the following two operations to adjust the MCD and PCD values:

- Recomputing the MCD values of vertices $w, x \in V$ for which $K(w)$ is updated and $(w, x) \in E$.

- Recomputing the PCD values of vertices $w, x, y \in V$ for which $K(w)$ is updated and $(w, x), (x, y) \in E$

Further shortcuts are possible, based on the K and MCD values of the updated vertices, to minimize the number of MCD and PCD re-computations. We defer the details to "Generic RCD Maintenance" Section.

Root Aware Edge Insertion

So far, in all our incremental algorithms, we first find a subgraph and its corresponding `cd` values by a BFS traversal (phase 1). In a second phase, we process that subgraph by reordering the vertices with respect to their `cd` values and remove the vertex with the minimum `cd` at each step. Traversing the subgraph and computing the `cd` values should be done prior to the second phase, since we need all the vertex degrees in the subgraph. However, Theorem 7 says that if the K value of some vertex changes, then the K value of at least one extremity of the inserted/removed edge, named as the root vertex (say u), must change. For the insertion algorithm, this fact suggests a *root-aware* approach, in which all vertices know whether the root still has a chance to change its K value. Additional operations are avoided once the algorithm detects that $PCD(u) \leq K(u)$, i.e., u cannot increment its K value. This condition implies that there is no chance for the root to increase its K value.

Algorithm 16: TRAVERSAL: insertEdge$(G(V,E), K(), u_1, u_2)$

Data: G: the graph, K: max-k values, MCD: max-core degrees, PCD: purecore degrees, (u_1,u_2): inserted edge

```
1  r ← u1                                                          ▷ Set the root
2  if K(u2) < K(u1) then r ← u2
3
4  G ← G ∪ (u1, u2)                                                ▷ Add the edge into G
5  prepareRCDs
   ▷ Perform a traversal over vertices that have root's K value, while evicting the ones
     that cannot be a part of a k+1-core
6  S ← empty stack                                                 ▷ To perform DFS
7  visited[v] = false, ∀v ∈ V                                      ▷ To perform DFS (lazy init)
8  evicted[v] = false, ∀v ∈ V                                      ▷ To remember evicted vert. (lazy init)
9  cd[v] = 0, ∀v ∈ V                                               ▷ To find vertices to be evicted (lazy init)
10 k ← K(r)                                                        ▷ Remember the K value of the root
11 cd[r] ← PCD(r)                                                  ▷ Set cd of root
12 S.push(r); visited[r] ← true
13 while not S.empty() do                                          ▷ Do a DFS traversal
14     v ← S.pop()
15     if cd[v] > k then                                           ▷ Vertex is currently part of a k+1-core
16         for each (v, w) ∈ E do
               ▷ Neighbouring vertex currently part of a k+1-core
17             if K(w) = k and MCD(w) > k and
18                                                  not visited[w] then
19                 S.push(w); visited[w] ← true
                   ▷ Use + as cd[w] may be < 0 due to evictions
20                 cd[w] ← cd[w] + PCD(w)
21     else                                                        ▷ Vertex cannot be part of a k+1-core
22         if not evicted[v] then                                  ▷ Recursively perform eviction
23             propagateEviction(G, K, cd, evicted, k, v)
24 for each v s.t. visited[v] do                                   ▷ Find visited vertices
25     if not evicted[v] then                                      ▷ If not evicted as well
26         K(v) ← K(v) + 1                                         ▷ The vertex is part of a k+1-core
27 recomputeRCDs
```

Algorithm 17: PROPAGATEEVICTION($G(V,E), K(), \texttt{cd}[], \texttt{evicted}[], k, v$)

```
Data: G: the graph, K: max-k values, cd: cd values, evicted: evicted values, k:
      max-k of root, v: evicted vertex
1 evicted[v] ← true
2 for each (v, w) ∈ E do
3     if K(w) = k then
4         cd[w] ← cd[w] − 1
5         if cd[w] = k and not evicted[w] then
6             PROPAGATEEVICTION(G, K, cd, evicted, k, v)
```

We realize this root-aware approach by applying a Depth-First Search (DFS) with an *eviction* mechanism, where the vertices $v \in V$ are evicted if $PCD(v) \leq K(v)$. By doing that, we combine phases 1 and 2.

The root-aware insertion procedure does not need the `cd` values of all the vertices in the subgraph. As a result, we create the `cd` values for each vertex *on-the-fly* during DFS, avoiding the first phase of our previous algorithms completely. We leverage the *residential core degrees*, introduced in Section 5.4.3, to speed up the creation of `cd` values. On-the-fly creation of `cd` values makes the insertion algorithm more efficient.

Algorithm 16 updates the K values of vertices by utilizing Algorithm 17, when edge (u, v) is inserted into the graph $G = (V, E)$. We start with preparing residential core degrees as explained in Section 5.4.3. Then we do a DFS starting from the root, say r, and at each step we pop the vertex v from the top of the stack and push some of its neighbors, say w, into the stack, if v and w are candidates to be in a $k+1$-core, where $k = K(r)$. If v cannot be in a $k+1$-core, then we mark it as evicted and initiate a recursive eviction from v. In a recursive eviction, the `cd` values of vertices x are

decremented, for $(v, x) \in E$ and $K(x) = k$. If the `cd` value of x turns out to be equal to k and x is not already marked as evicted, then we start another eviction from x. When DFS finishes, we increment the K values of all vertices that were visited but not evicted. Last, we adjust the residential core degrees as discussed in Section 5.4.3.

Theorem 9. *Algorithm 16 finds the vertices whose K values needs to be updated.*

Proof. First, we prove that after an edge is inserted, if $PCD(u) \leq K(u)$ for a vertex $u \in V$, then it cannot increase its K value as shown in lines labeled 15 and 5 in Algorithms 16 and 17, respectively. Assume it does and say that $k = K(u)$. Then, after $K(u)$ increases, u must have at least $k + 1$ neighbors with greater or equal K value, by Observation 5. However, at most k neighbors of u can have their K values greater than or equal to k after $K(u)$ increases, since $PCD(u) \leq K(u)$ before $K(u)$ is increased, i.e., a contradiction.

Second, we prove that if $PCD(u) \leq K(u)$, where u is the visited vertex, then $PCD(w)$ must be decremented as shown in line labeled 4 in Algorithm 17, where w is a neighbor of u having K value of $K(u)$. Assume that $PCD(w)$ is not decremented. Then u is supposed to be in the max-core of w, if w increases its K value. However, u cannot be in the max-core of w, since it cannot increase its K value as proved in the first paragraph of proof, contradiction.

We traverse the graph starting from the root and evict the vertices as shown in above proofs. Non-evicted and traversed vertices increment their K values at the end of the algorithm. □

Edge Removal

Edge removal using the traversal algorithm employs a similar on-the-fly updating of the `cd` values. A key difference from the edge insertion algorithm is that, the edge removal relies on a simple recursion on the vertices whose K values should be decremented.

The traversal algorithm for edge removal is presented in Algorithm 18. We start with preparing residential core degrees as explained in Section 5.4.3. Depending on

the equality of K values of the edge extremities, i.e., u_1 and u_2, we apply one or two recursive propagation operations to correctly calculate the K values. In the propagation operation, if the `cd` value of v turns out to be below its K value (i.e., K needs to be decremented), we perform a recursive *dismissal* operation starting from v, which is given in Algorithm 19. In the recursive dismissal operation, we decrement $K(v)$ and the `cd` values of vertices w, where $(v, w) \in E$, $K(w) = k$, and k is the K value of the root. If w gets a smaller `cd` value than k and $K(w)$ has not decremented yet, then we start another recursive dismissal, but this time from w. When the recursion completes, we adjust the residential core degrees as discussed in Section 5.4.3.

5.4.4 Generic Multihop Traversal Algorithm for Insertion

The traversal algorithm that handles edge insertions, presented in Section 5.4.3, makes use of the MCD and PCD values of the vertices. MCD value of a vertex contains information from the 1-hop neighborhood, whereas PCD value contains information from the 2-hop neighborhood. However, the traversal algorithm can be generalized to utilize multihop information (greater than 2-hops). Higher hop counts enable faster detection of vertices that cannot appear in a larger core, yet increase the time spent to maintain the residential core degrees. As such, it involves a trade-off. Yet, in order to investigate this trade-off, we need to support using information from arbitrary number of hops. Accordingly, in this section, we present the generic traversal

Algorithm 18: TRAVERSAL: removeEdge$(G(V,E), K(), u_1, u_2)$

```
Data: G: the graph, K: max-k values, MCD: max-core degrees, PCD: purecore
      degrees, (u1,u2): removed edge
r ← u1                                                    ▷ Set the root
if K(u2) < K(u1) then r ← u2

G ← G \ (u1, u2)                                  ▷ Remove the edge from G
Prepare RCDs
  ▷ Perform a DFS traversal over vertices that have root's K value, while dismissing
    the ones that cannot be a part of a k-1-core
visited[v] = false, ∀v ∈ V                          ▷ To perform DFS (lazy init)
dismissed[v] = false, ∀v ∈ V                  ▷ To remember dis. vertices (lazy init)
cd[v] = 0, ∀v ∈ V                  ▷ To find vertices to be dismissed (lazy init)
k ← K(r)                                  ▷ Remember the K value of the root
if K(u1) ≠ K(u2) then
    visited[r] ← true
    cd[r] ← MCD(r)
    if cd[r] < k then
        propagateDismissal(G, K, MCD, cd, dismissed,
        visited, k, r)
else
    visited[u1] ← true
    cd[u1] ← MCD(u1)
    if cd[u1] < k then
        propagateDismissal(G, K, MCD, cd, dismissed,
        visited, k, u1)
    visited[u2] ← true
    cd[u2] ← MCD(u2)
    if not dismissed[u2] and cd[u2] < k then
        propagateDismissal(G, K, MCD, cd, dismissed,
        visited, k, u2)
Recompute RCDs
```

algorithm for edge insertion, which leverages the multihop residential information of vertices for faster calculation of K values.

First, we present the generic definition for n-hop residential core degrees.

Algorithm 19: PROPAGATEDISMISSAL($G(V,E), K(), MCD(), \texttt{cd}, \texttt{dismissed}, \texttt{visited}, k, v$)

Data: G: the graph, K: max-k values, MCD: max core degrees, `cd`: cd values, `dismissed`: dismissed values, `visited`: visited values, k: max-k of root, v: dismissed vertex

```
1  dismissed[v] ← true
2  K(v) ← K(v) − 1                                  ▷ The vertex is part of a k-1-core
3  for each (v, w) ∈ E do
4      if K(w) = k then
5          if not visited[w] then
6              cd[w] ← cd[w] + MCD(w)
7              visited[v] ← true
8          cd[w] ← cd[w] − 1
9          if cd[w] < k and not dismissed[w] then
10             PROPAGATEDISMISSAL(G, K, MCD, cd, dismissed, visited, k, w)
```

Definition 9. *The n-core degree of a vertex u, denoted as $RCD(u,n)$ where $n \geq 0$, is defined in terms of the number of u's neighbors, w, such that **either** $K(u) = K(w)$ and $RCD(w, n-1) > K(u)$ **or** $K(u) < K(w)$. When $n = 0$, $RCD(.,n)$ of a vertex is defined to be ∞.*

For a vertex v, its n-core degree is defined recursively. It is simply the number of neighbors w it has with either higher K value than v's K value or has equal K value and higher $(n-1)$-core degree than v's K value. With higher values of n, $RCD(.,n)$ value of a vertex becomes a stronger indicator of eligibility to increase its K value. Value of n implies the extent of neighborhood information being used. For $n = 1$, only the information on 1-hop neighbors are used, and for $n = 2$, the information on hop-1 and hop-2 neighbors are utilized. Note that, when $n = 1$, $RCD(.,n)$ definition reduces to MCD (maximum-core degree), given in Definition 6. Also, when $n = 2$, it has the same definition as the PCD (pure-core degree), given in Definition 8.

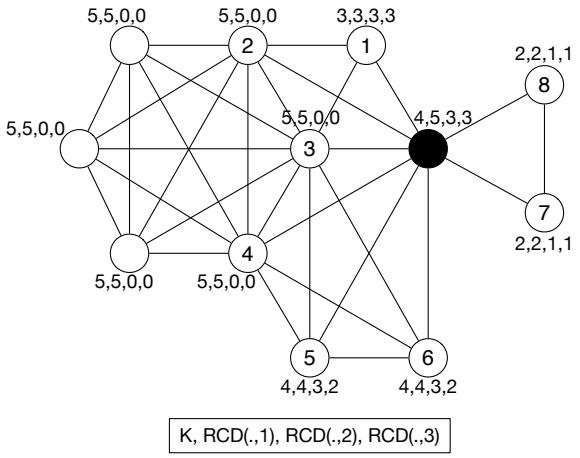


Figure 5.2: Illustration of RCD values of the vertices in the sample graph

Figure 5.2 shows an example graph to illustrate the $RCD(u, n)$ definition. K, $RCD(., 1)$ (MCD), $RCD(., 2)$ (PCD) and $RCD(., 3)$ values of vertices are shown next to them. For example, the $RCD(., 3)$ value of the black vertex is computed as follows: there are 3 neighbors (vertices 2, 3, and 4) with a K value of 5, which is greater than the K value of black vertex. The $RCD(., 3)$ value is then incremented by 3. Vertices 1, 7, and 8 have smaller K values than the black vertex, thus they do are not counted. The K value of vertices 5 and 6 are equal to the K value of the black vertex, and therefore we check if their $RCD(., 2)$ values are greater than their K values. However, it is not the case, since $RCD(., 2)$ value of both vertices is 3 and the K value of both vertices is 4. As a result, $RCD(., 3)$ value of black vertex is set to 3.

The generalized traversal algorithm for insertion, which utilizes the multihop information based on a given hop distance n (where $n > 1$), is given in Algorithm 20. The main difference in the multihop traversal algorithm is that, we use $RCD(., n)$ values instead of PCD values and $RCD(., n-1)$ values in place of MCD values. Notice that we use the same PROPAGATEEVICTION procedure (Algorithm 17) in the multihop traversal algorithm as well. Differences between Algorithm 16 and Algorithm 20 are highlighted on lines 6, 13, 20, 24, and 34.

Lines 6 and 34 together represent the generalized version of RCD maintenance for multihop residential core degrees, and will be explained in detail in "Generic RCD Maintenance" Section. In Lines 13, 20, and 24, RCD values of hop n are used to reduce the traversal space. As stated earlier, RCDs with increasing hop values become stronger indicators of whether the K value of a vertex will increase or not. We expect that the traversal algorithm will explore a smaller space with higher n values. However, higher n values come with increasing RCD maintenance costs. We experimentally evaluated our multihop traversal algorithm with different n values to find the optimal hop value. As we will discuss later in Section 5.6.5, the optimal value changes based on the dataset.

Generic RCD Maintenance

As stated earlier in Section 5.4.3, maintaining RCD values is a non-trivial operation. Yet it is critical in reducing the scopes of the traversals, potentially bringing down the cost of edge modifications. Overall, efficient mechanisms for maintaining

RCD values is needed. Here, we introduce the generic versions of the RCD maintenance algorithms, which update the RCD values of vertices up to the given hop count n. In other words, given the number of hops, n, the proposed algorithms maintain the RCD values for n, $n-1$, $\cdots$, and 1.

MULTIHOPPREPARERCDSINSERTION method given in Algorithm 21 is used in Line 6 of the multihop traversal based edge insertion algorithm given in Algorithm 20. It prepares the RCD values before the multihop traversal operation for the given inserted edge, (u_1, u_2), and hop distance, n. This preparation is needed as the RCD values of the root(s) may have changed due to the updated degrees and this change may have propagated to RCD values of other vertices. The preparation phase is performed assuming that the K values are intact. Those will be updated during the traversal, and a re-computation of RCDs would be required at the end (MULTIHOPRECOMPUTERCDS procedure).

The preparation starts with determining the *root* vertices based on their K values. If the K values of the extremities of the inserted edge are not equal, we increment the $RCD(r, h)$ value of root for all $h \leq n$. The rationale behind this is that, the root vertex gains a new neighbor with a higher K value and by Definition 9 it increases all RCD values of root by one. Following this increment operation, we check if the $RCD(r, h)$ has exceeded k, because this implies further changes in $RCD(., h+1)$ values of r's neighbor vertices (by Definition 9). In the preparation phase, $RCD(., n)$ of a vertex only changes when $RCD(., n-1)$ of a neighbor changes and that is what we

are checking for. Remember that $RCD(u, n)$ is the number of u's neighbors, w, where either $K(u) < K(w)$ or $K(u) = K(w)$ and $RCD(w, n-1) > K(u)$. Throughout the algorithm, we accumulate the vertices whose $RCD(., h)$ values just exceed k in the next $frontiers$ set, where h is the hop number. We avoid this accumulation operation if the last hop number h is being processed, since there is no need for further processing in that case. When the hop number h is greater than 1, we process the neighbors (with the same K value) of the vertices in the current $frontiers$ set by incrementing their $RCD(., h)$ values. We also perform checks to see if k is exceeded and accordingly populate the next $frontiers$ set.

If the K values of the extremities of the inserted edge are equal, we do different operations for $h = 1$ and $h > 1$, where h is the current hop number. For $h = 1$, where $RCD(u, 1)$ is actually equal to $MCD(u)$, we just increment the $RCD(., 1)$ values of both extremities of the inserted edge (by Definition 9) and perform checks to see if k is exceeded and accordingly populate the next $frontiers$ set. If $h > 1$, we need to handle the new inserted edge separately. Let us say u_1 and u_2 are the extremities of the inserted edge. We first check if the $RCD(u_1, h-1)$ (and dually $RCD(u_2, h-1)$) is greater than k. If so, we increment the $RCD(u_2, h)$ (and dually $RCD(u_1, h)$) and perform the k value checks to populate the next frontier as needed. After that, we process the neighbors (with the same K value) of the vertices in current $frontiers$ set. One important difference in this step is that we exclude the edge between u_1 and u_2, because that edge is already handled.

Multihop algorithms are only applicable for the edge insertion operation. For removal, using 1-hop information (MCD values) is necessary and sufficient. Therefore, going for multihop information does not bring any additional benefit in terms of the running time. However, given that we are interested in sliding window scenarios, where removals happen together with insertions, we need to accommodate the maintenance of RCD values when there is an edge removal. For this purpose, we develop the MULTIHOPPREPARERCDSREMOVAL method. Algorithm 22 adjusts the RCD values when there is an edge removal and very similar to Algorithm 21. One important difference is that, instead of incrementing the RCD values, we need to decrement them whenever necessary. We also check if the $RCD(., h)$ value goes below $k+1$, which implies changes in $RCD(., h+1)$ values of neighbor vertices. Another difference between Algorithm 22 and Algorithm 21 exists when the K values of the removed edge extremities are equal. In this case, we need to remember the RCD values for all hop numbers before the edge removal operation. This enables us to process the hop numbers $h > 1$.

After the multihop traversal, if the K values of some vertices are incremented, then this will create a cascading effect on RCD values of the vertices around. Efficiently handling the cascades and doing the update operations is again of great importance. Algorithm 23 finds those vertices whose RCD values need to be updated and efficiently updates these RCD values. It has two main parameters: the set of vertices whose K values are updated (`changed`), and the hop distance until which RCD values

are to be updated (n). We start the algorithm by marking the `changed` vertices as *visited*. Throughout the algorithm, we mark the vertices via the visited array to prevent duplicates during the update procedure. In the main for loop (the second one), we process the updates for each hop, in order. At each iteration, we populate the `changed` set with the updated vertices and then update the RCD values of the vertices in `changed`. Cascading effect propagates by a single hop neighborhood at each iteration. In other words, if we assume that a vertex u has its K value updated and we want 3-hop distance RCD values to be updated; $RCD(1)$, $RCD(2)$ and $RCD(3)$ of u will be updated. Furthermore, $RCD(2)$ and $RCD(3)$ values of *some* vertices in u's hop-1 neighborhood will be updated, and $RCD(3)$ values of *some* vertices in u's hop-2 neighborhood will be updated.

Pruning the vertices in the neighborhood is critical in making the procedure efficient. For the edge insertion case, given vertex v, we prune the neighborhood vertices by checking if they are visited previously and if the K value of the neighbor vertex is either equal to K value of v or equal to K value of v minus 1 (plus 1 for the edge removal case). The reason behind this check is based on Definition 9. $RCD(n)$ of a neighbor vertex may change iff the K values are equal. For the edge insertion case, given that there are also some vertices whose K values are incremented, we need to consider them as well by checking the neighbor vertices with one less K value. Likewise, for the edge removal, we check the neighbor vertices with one more K value, as

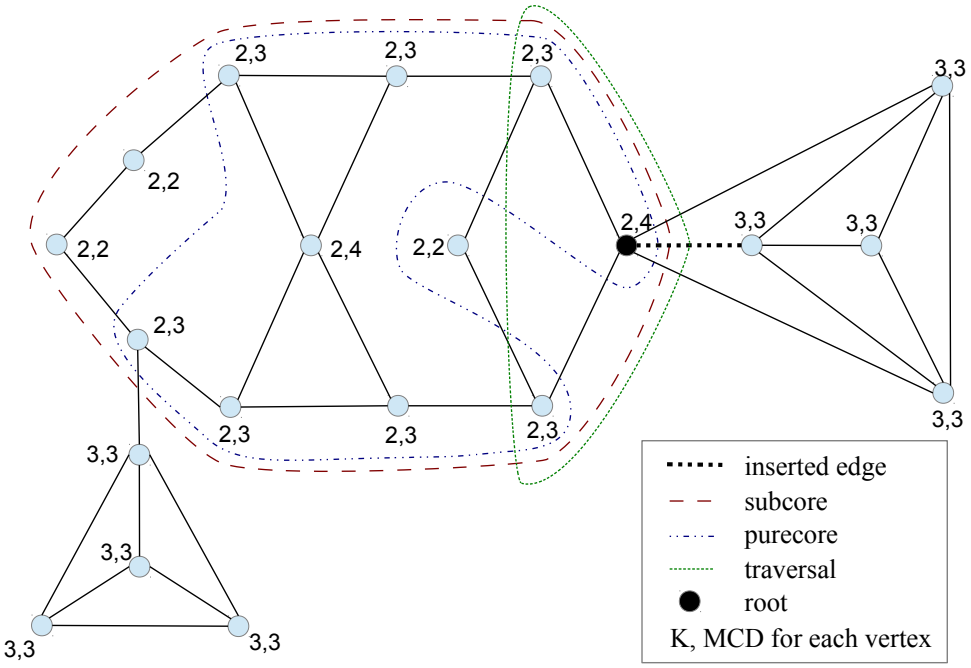


Figure 5.3: Illustration of the vertices visited by the subcore, purecore, and the traversal algorithms.

stated with comments in the pseudocode of Algorithm 23. We accumulate the vertices to be updated in `changed` set and update their RCD values for the hop distance at that iteration. In summary, we handle the cascading effect of RCD maintenance efficiently by the aforementioned pruning techniques.

5.4.5 Illustrative Example

Figure 5.3 illustrates the subcore, purecore, and traversal algorithms using a sample graph. The edge drawn using a dashed bold line is the one that is being inserted into

the graph. The vertex shown in black is the root vertex. The graph shows the K values and the MCD values for each vertex before the insertion. The set of vertices visited by each one of the subcore, purecore, and the traversal algorithms, for the purpose of updating the K values, is shown in the figure. The subcore algorithm visits the vertices with K value of 2, which are reachable from the root. The purecore algorithm visits the vertices with K value of 2 and MCD value of greater than 2 that are reachable from the root.

The traversal algorithm starts by updating the MCD value of the root to 5, due to the new edge. Then, DFS starts and pushes the root to the stack. When the root is popped from the stack, its two neighbors with (K, MCD) values of (2, 3) are pushed to the stack (MCD values greater than K value of the root, indicating that they can potentially be part of a larger core). Say that those vertices are x at the top and y at the bottom in Figure 5.3. Based on Definition 8, the `cd` values of x and y are updated to 2 since their PCD values are 2. After that, we move to the next iteration, and pop vertex x from the stack. The `cd` value of x is 2, which is not greater than the K value of the root. This means that it cannot participate in a higher core. As a result, no neighbors of x are visited and PROPAGATEEVICTION is initiated for x. In PROPAGATEEVICTION, x is evicted and the `cd` values of all neighbors of x are decremented, since all neighbors have a K value of 2 (same as root). Furthermore, PROPAGATEEVICTION is not initiated for any neighbor of x, since the `cd` value of the

root (one of x's neighbors) becomes 4, and the `cd` value of other two neighbors of x become -1, all of which are different than the K value of the root.

In the next step, the DFS pops vertex y from the stack. Similar to x, the `cd` value of y is 2, which is not greater than the K value of the root. As a result, no neighbors of y are visited and PROPAGATEEVICTION is initiated for y. In PROPAGATEEVICTION, y is evicted and the `cd` values of all neighbors of y are decremented, since they have a K value of 2 (same as root). Furthermore, PROPAGATEEVICTION is not initiated for any neighbor of y, since the `cd` value of y's neighbors differ from the K value of the root. After these operations, the stack is empty, and the only vertex that is visited but not evicted is the root. As a result, the K value of the root is incremented. As the last step, the MCD and PCD values of vertices are updated as explained in Section 5.4.3.

We can easily see that the set of vertices visited by the subcore algorithm is larger than that of the purecore algorithm, whereas the traversal algorithm visits the smallest number of vertices compared to the other two.

5.5 Implementation

In this section we provide details about efficient implementations of the incremental algorithms presented. In particular, we discuss two main issues: the lazy initialization of arrays used in the algorithms, and the repeated sorting of the `cd` arrays.

5.5.1 Lazy arrays

The non-incremental algorithms for computing the k-core decomposition perform work that is proportional to the size of the graph. As a result, our incremental algorithms should avoid any operation that requires work in the order of the size of the graph. However, several of our algorithms include arrays like `visited`, `evicted`, `cd`, etc., that are initialized to a default value and accessed using vertex indices. For these, we use lazy arrays to avoid allocations and initializations in the order of the graph size.

A lazy array employs a hash map based data structure to implement a sparse array. For a given vertex, if its value is not currently being stored in the hash map, it is assumed to have the designated default value. When a different value for the vertex needs to be stored, the entry for it is created in the hash map.

Since hash maps provide constant lookup time, using lazy arrays achieves significant speedup when the number of vertices visited by the incremental algorithms is smaller than the graph size. On the other hand, when the number of vertices visited gets large, relative to the graph size, lazy arrays start performing worse, since the constant overhead of accessing a data item in a hash map is significantly higher than that of regular arrays.

Given that our algorithms locate a small subset of vertices for updating the k-core decomposition of a graph, the use of lazy arrays is almost always beneficial. For graphs that have very large max-k cores, relative to the graph size, (which we show

to be an uncommon occurrence in practice) an implementation of lazy arrays that switches to a dense representation when the occupation percentage of the array gets larger can be an effective solution, even though we do not implement that variation in this study.

5.5.2 Bucket sort

Several of our algorithms require reordering the set of unprocessed vertices in a subgraph (such as a subcore or a purecore) based on their `cd` values. In the worst case this subgraph could be as large as the graph itself (again, this is uncommon in real-world graphs). To perform this re-sorting efficiently, we use bucket sort. Note that the `cd` values have a very small range, and thus bucket sort not only provides $O(N)$ sort time for the initial sort (where N is the subcore or purecore size), but it also enables $O(1)$ updates when a vertex changes its `cd` value (in our case the values only decrease). We use a bucket data structure that relies on linked lists for storing its bucket contents and on a hash map to quickly locate the link list entry of any given vertex.

5.6 Experimental Evaluation

In this section, we evaluate how the proposed algorithms behave under different scenarios. The first set of experiments shows the scalability of our best performing algorithm by studying its runtime performance as the size of the synthetic datasets

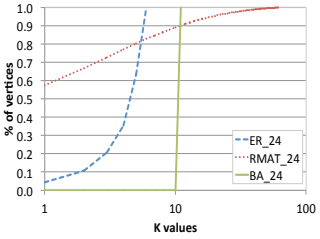


Figure 5.4: Cumulative K value distribution for synthetic graphs.

Graph file	Number of vertices	Number of edges	Maximum degree	Average degree	Max k
caidaRouterLevel	192,244	609,066	1,071	6.336	32
eu-2005	862,664	16,138,468	68,963	37.415	388
citationCiteseer	268,495	1,156,647	1,318	8.616	15
coAuthorsCiteseer	227,320	814,134	1,372	7.163	86
coAuthorsDBLP	299,067	977,676	336	6.538	114
coPapersCiteseer	434,102	16,036,720	1,188	73.885	844
cond-mat	16,726	47,594	107	5.691	17
power	4,941	6,594	19	2.669	5
protein-interaction-1	9,673	37,081	270	7.667	14

Table 5.1: Real-world graph datasets and their properties.

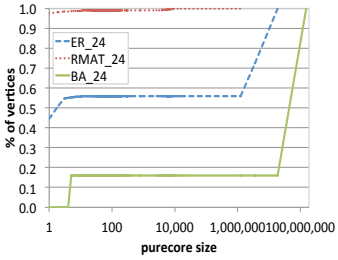


Figure 5.5: Cumulative purecore size distribution for synthetic graphs.

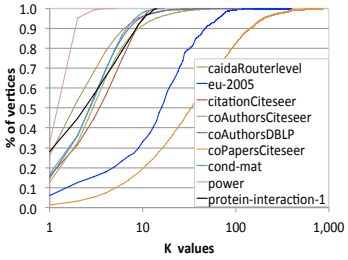


Figure 5.6: Cumulative K value distribution for real-world graphs.

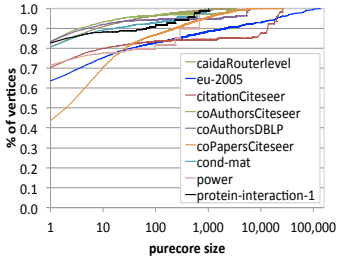


Figure 5.7: Cumulative purecore size distribution for real-world graphs.

increases. The second set of experiments compare the performance of our incremental algorithms with respect to each other on real datasets. The third experiment investigates the performance variation depending on the K values of u and v, when an edge (u, v) is inserted/removed. The last set of experiments examine the performance tradeoffs associated with the multihop traversal insertion and generic RCD maintenance algorithms, presented in Section 5.4.4.

Our algorithms are implemented in C++ and compiled with `gcc 4.4.4` at -O2 optimization level. All experiments are executed sequentially on a Linux operating system running on a machine with two Intel Xeon E5520 2.27GHz CPUs, with 48GB of RAM.

5.6.1 Datasets

Our dataset includes synthetic and real graphs. For synthetic graphs, we use the SNAP library [166] to generate networks following three different models. The first is the Erdös-Renyi model, which generates random graphs [59]. We used $p = 0.1$ to put an edge among two specified vertices and we specify $|E|/|V|$ as 8. The second is the Barabasi-Albert preferential attachment model [20], which follows a power law for the vertex degree distributions. We configure it such that each new vertex added by the generation algorithm creates 11 edges. The third model, generated with SNAP's R-MAT generator [39], follows a power law vertex degree distribution and also exhibits small world properties. We set the partition probabilities as $[0.45, 0.25, 0.20, 0.10]$, to approximate the k-core distribution of real citation graphs in our dataset.

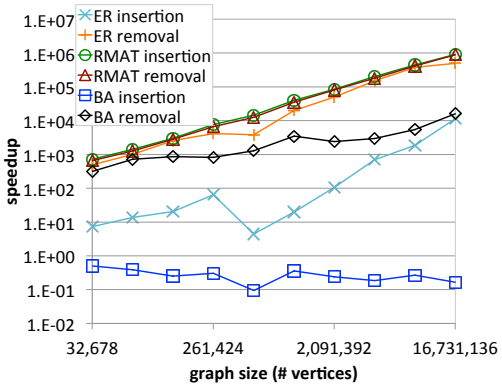


Figure 5.8: Speedup of incremental insertion and removal algorithms for synthetic graphs when varying the graph size from 2^{15} to 2^{24}. Removal scales better than insertion, reaching around 10^6 speedup.

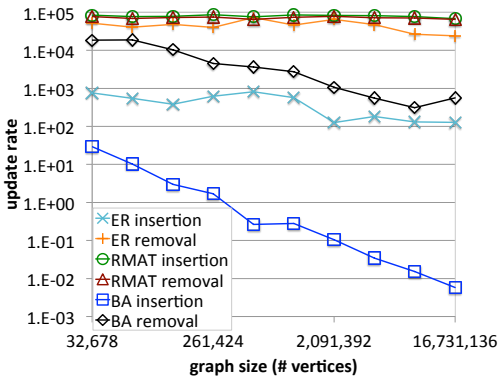


Figure 5.9: Update rates of incremental insertion and removal algorithms for synthetic graphs when varying the graph size from 2^{15} to 2^{24}.

Figures 5.4 and 5.5 show the cumulative distribution of K values and purecore sizes (i.e., number of edges of the purecore subgraph of each vertex in the graph) for the synthetic datasets with 2^{24} vertices. For a graph $G = (V, E)$, we calculate the purecore of each vertex $u \in V$ by using Algorithm 15. These figures reveal the structure of the generated graphs and how it impacts the incremental k-core decomposition performance. The K value distribution is an indication of the connectivity of the graph, while the purecore size is an indication of the potential runtime of our incremental algorithms when an edge incident upon a given vertex is inserted/removed.

As shown in Figure 5.4, the graph based on the Barabasi-Albert model (BA_24) has 100% of its vertices with $K = 11$. In addition, over 80% of its vertices result in a purecore size of over 100 million vertices. These properties of the BA graphs

are due to the graph generation algorithm of the BA model, where newly inserted edges are likely to connect high degree vertices. As we will see shortly, real-world graphs do not follow such properties and the figure shows that the BA model is very poor in approximating real world graphs in terms of the K value distribution. The RMAT generated graph (RMAT_24) has nearly 60% of its vertices with very low K values. As the K value increases, the percentage of vertices with that K value decreases. Furthermore, 98% of its vertices have very small purecore sizes. The ER generated graph (ER_24) has K values up to 6, and as the K value increases, the percentage of vertices with that K value also increases. The latter behavior is unlike the RMAT generated graph. As we will see shortly, most real-world graphs of interest behave more closely to the RMAT generated graphs with respect to their K value distribution.

The real graphs we use are from the 10th DIMACS Graph Partitioning and Graph Clustering Implementation Challenge repository [51] and include internet router level and European domain computer network graphs (caidaRouterLevel and eu-2005), co-author and citation network graphs (citationCiteseer, coAuthorsCiteseer, coAuthors-DBLP, coPapersCiteseer), condensed matter collaboration network graphs (cond-mat), power grid network graphs (power), and protein interaction network graphs (protein-interaction-1). Table 5.1 provides the details about each used graph, including their vertex and edge set size, maximum and average degrees, and their maximum k value. All graphs are undirected.

Figure 5.6 shows the K value distribution for all graphs in Table 5.1. The figure shows that the vertices of both coPapersCiteseer and eu-2005 have highly variant K values. Figure 5.7 shows the purecore size distribution for our real datasets. The data indicates that all of the graphs have at least 80% of their vertices with corresponding purecore sizes of less than 100. This is an indication that our incremental algorithms are expected to perform well on these graphs.

As all our graphs are originally static, we emulate a streaming algorithm by considering that the whole set of edges and vertices constitute a *sliding window* snapshot. For evaluating algorithm execution, we first evict a random edge from the current graph in the window. This emulates the behavior of a full sliding window, which must open space for inserting a new data item. We then insert a new edge between two random vertices. We also evaluate worst case execution times by inserting and removing edges from vertices that have top *purecore* sizes. Such results are similar to the random insertion case, and are omitted for brevity. Note that we do not assume any specific data distribution with respect to which edges get inserted or deleted. In addition, we make no assumptions regarding edge arrival rates. Instead, we evaluate the performance of our algorithms' processing updates as fast as possible.

5.6.2 Scalability

In this experiment, we evaluate the performance of the traversal algorithm (Section 5.4.3) as the size of the synthetic graphs increase. We first report speedup numbers, which are obtained by comparing the traversal runtimes with our baseline

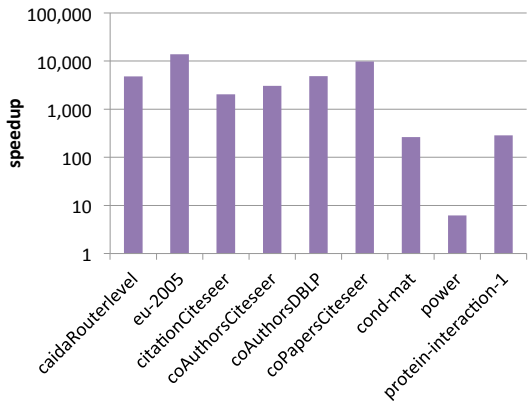


Figure 5.10: Subcore algorithm speedups for real datasets when compared to the baseline. Our incremental algorithm runs up to $14,000\times$ faster than the non-incremental algorithm.

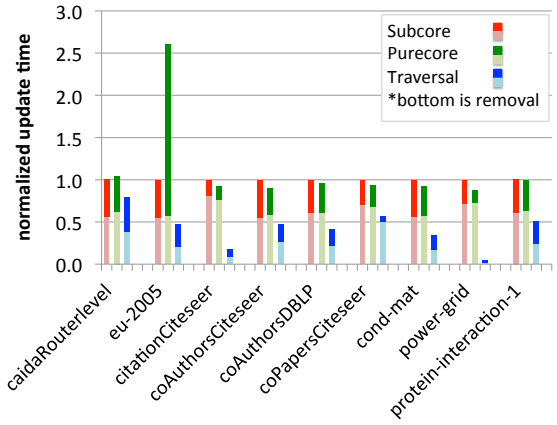


Figure 5.11: Average update time comparison of incremental algorithms when processing real datasets. Times are normalized by the average update time of the subcore algorithm. Traversal algorithm shows the best performance for all datasets.

— the non-incremental version of k-core decomposition (Algorithm 11), then present the update rates, which show the number of edge removals/insertions processed per second. Testing the algorithm under different graph sizes emulates the scenario where a streaming algorithm uses different sliding window sizes.

Figure 5.8 shows the speedup of our incremental insertion and removal algorithms when the number of vertices from the graph range from 2^{15} to 2^{24}. For the insertion algorithm, the RMAT graph shows the best scalability, with speedups ranging from $717\times$ to $920,000\times$ (almost 6 order of magnitude). This drastic speedup is because the K values of the vertices in the graph have high variability and majority of the vertices

have very small purecore sizes, as shown in Figures 5.4 and 5.5 for the RMAT graph with size 2^{24}. Such factors result in very fast insertions. The insertion of edges into the graph following the Erdös-Renyi model (ER) show speedup ranging from 4.43× to $11,500\times$. Although it also scales well with the size of the graph, the speedups are not as high as the ones observed for the RMAT graph. This behavior can be explained by the fact that the ER graph has a more uniform K value distribution when compared to the RMAT graph. Furthermore, when the graph has size of 2^{24}, over 40% of its insertions may result in touching purecores of over 1 million edges. When inserting edges into graph based on the Barabasi-Albert model (BA), our incremental algorithm is worse than the non-incremental one. As we discussed earlier, in these graphs all vertices have the same K value initially, resulting in subcore sizes that are almost equal to the graph size. In this case, the incremental algorithm does not provide any benefit on top of the base one, yet brings additional computation overheads (such as due to lazy arrays). As we will show shortly, this nature of the BA graphs are not found in real world graphs.

The removal algorithm scales for all three synthetic graphs, where the speedup ranges from 675× to $1,321,200\times$. For the ER and BA graphs, the removal algorithm scales better than the insertion one because it has much lower cost (see Section 5.4.3). At large scales, we notice that the use of incremental algorithms becomes even more critical, since the cost of the baseline is linear in the size of the graph.

The scalability experiments indicate how good our incremental algorithm can perform for different graph sizes when there are k-core decomposition queries (*read queries*) interspersed with edge insertion and removal (*write queries*). Taking the RMAT graph with size of 2^{24} vertices as an example, we can see that if the write/read ratio is less than $1,972,945$ (the average speedup of one removal and one insertion), it is better to use the incremental algorithm than to compute the k-core decomposition from scratch after inserting new edges and removing the oldest ones from the graph (sliding window scenario).

Figure 5.9 shows the update rates, i.e., number of edges processed per second, for our incremental insertion and removal algorithms when the number of vertices in the graph ranges from 2^{15} to 2^{24}. For RMAT graphs, both insertion and removal rates reach up to 205,000 and 87,000 updates/sec and, more importantly, update rates do not change when the graph size increases. ER graphs have lower update rates for both insertion and removal. Removal rates for ER graphs stay stable as the graph size increases and insertion rates only decrease by a factor of 3 (from 4,478 to 1,867) when the graph size increases from 2^{15} to 2^{24}. For BA graphs, update rates for removal decreases from 25,688 to 15,147 when the graph size increases. Insertion rate has a similar decreasing behavior with the graph size. However, the rates are much lower — starting from 108 and decreasing to 0.7 when the graph size gets bigger. The decreasing trend for the BA graphs is due to the large subcore sizes that are

proportional to the graph size. Again, we will show that real world graphs do not exhibit this behavior.

5.6.3 Performance comparison

In this experiment, we analyze how our three incremental algorithms perform when processing one edge removal and one edge insertion (i.e., one sliding window operation) on the real datasets described in Table 5.1. This helps us to see whether the algorithm that is expected to give the best results, Traversal, shows the best performance for all the real datasets we have.

Figure 5.10 shows the performance of the subcore algorithm (Section 5.4.1) considering the average time taken by one graph update. The performance is shown in terms of the speedup provided by the incremental algorithm compared to the non-incremental one. The speedups vary from 6.2× to 14,000×. The datasets in which the incremental algorithm performs the best are the eu-2005 and coPapersCiteseer. Similar to the results obtained in the synthetic graphs, the performance of the subcore algorithm benefits from the high variability in the K value distribution of the graph. The dataset in which the subcore algorithm performs the worst is power. This is because 63.19% of the vertices in the power graph have the same K value, yielding large subcore sizes.

Figure 5.11 shows the average update time of each algorithm normalized by the update time of the subcore algorithm. Each group of 3 columns shows the results for a given dataset. For each group, the results are displayed in the following order:

subcore (Section 5.4.1), purecore (Section 5.4.2), and traversal (with residential core degrees) (Section 5.4.3). The stacked columns represent the update time attributed to the removal (bottom) and insertion operations (top).

The results show that the purecore algorithm can perform worse than the subcore one for some datasets (caidaRouterlevel and eu-2005) even though the purecore of a vertex is always smaller than or equal to the subcore of a vertex. This is due to the additional work performed to locate a smaller subgraph. This additional work is not always worth it if the purecore is not sufficiently small compared to the subcore. The figure also displays that the traversal algorithm shows the best performance for all datasets, being up to 20× better than the subcore algorithm. The traversal algorithm shows dramatic improvement compared to subcore when processing citationCiteseer and power graphs. Our results also show that the traversal algorithm has the most efficient removal for all datasets.

We also investigate the impact of Residential Core Degrees on synthetic graphs generated using the Erdös-Renyi model. Table 5.2 shows the average time in seconds spent for one edge removal plus one edge insertion with the traversal algorithm. For each graph, we ran the traversal algorithm with and without the Residential Core Degrees. The results show that using Residential Core Degrees provides up to %48 less runtime. The results for RMAT, not included here for brevity, show less improvement.

5.6.4 Performance variation

In this section, we evaluate the performance of the traversal algorithm when inserting and removing random edges into vertices with varying K values. The objective is to understand how the execution time varies as edge insertions and removals are performed on different parts of the graph with different connectivity characteristics. For instance, performance implications of adding an edge between a vertex that has a high K value and one with a low K value versus between two vertices having close K values.

Figure 5.12 shows the performance results for the citationCiteseer graph, which serves as a good representative for our real dataset. This graph has vertices with K values varying from 1 to 15. A bubble in the graph indicates the time taken to insert or remove an edge between two random vertices u and v. If $K(u) \leq K(v)$, $K(u)$ is displayed on the x-axis, while $K(v)$ is displayed on the y-axis. The size of the bubble indicates the average execution time for the insertion (pink) and removal (green) of an edge. The larger the bubble is, the greater the execution time is.

The graph shows that the runtime of the traversal algorithm has low variability. This is a good property, as it means that the algorithm is able to locate a small subgraph to traverse irrespective of the properties of the neighborhoods of the two vertices u and v. Our algorithm shows low runtime variability, as we consistently traverse subgraphs using the vertex with the lowest K value as root.

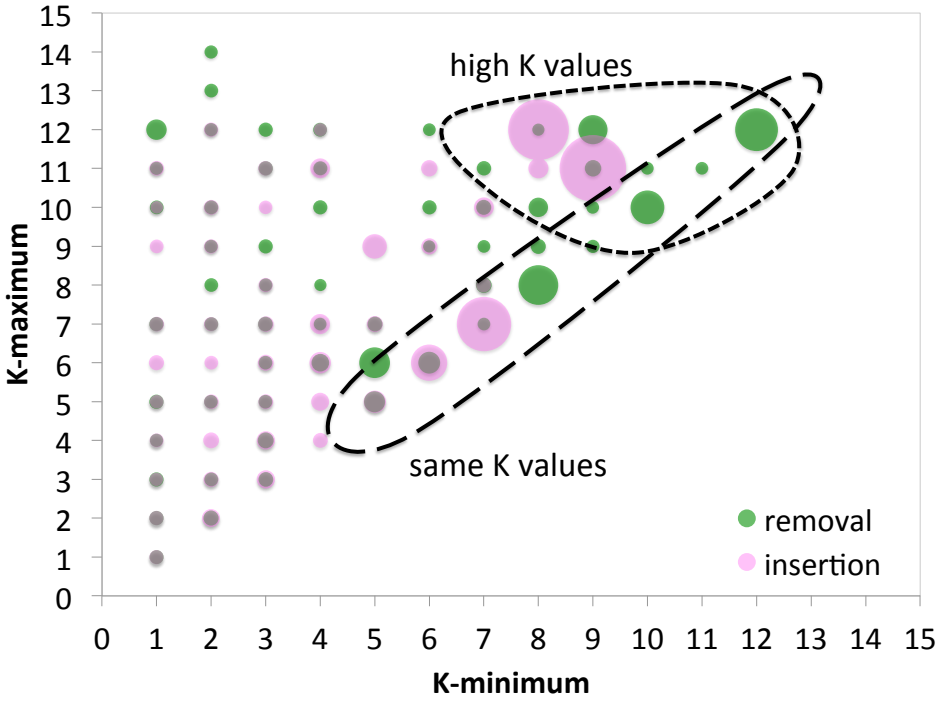


Figure 5.12: Edge insertion and removal execution times of the traversal algorithm for different K values. Runtime shows low variability when changing parts of the graph with different connectivity characteristics.

Execution times vary more when the K values of the different vertices are the same (diagonal). The reason is that the traversal algorithm visits the subgraphs associated with both vertices affected by the new edge, resulting in longer execution times. We also see that insertions between vertices with large K values have large execution times. In general, the execution times we see are proportional to the sizes of the subcores and **not** to the max-cores. In other words, what affects the execution time are the sizes of the subgraphs with the same K value. For small K values, such

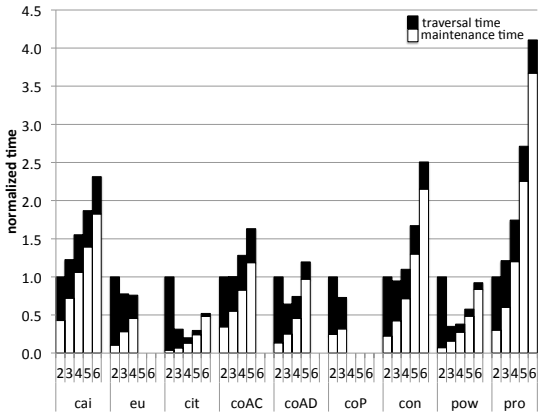


Figure 5.13: Maintenance times increase with the higher hop counts, yet the traversal times decrease in general. When the running time of the 2-hop variant is dominated by the traversal time, increasing hop counts bring significant improvement in terms of the traversal times. 3-hop and 4-hop variants are shown to give the best overall performance for 5 of the graphs, out of 9 total.

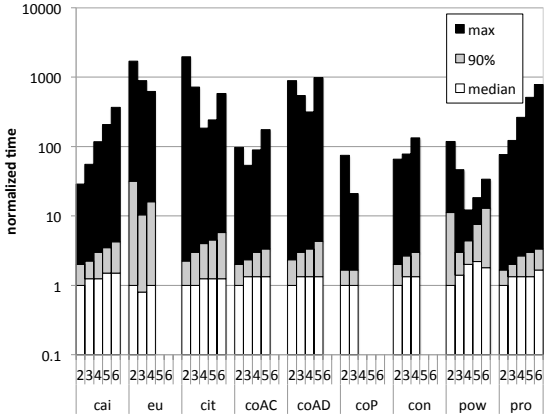


Figure 5.14: Detailed running time comparison for varying hop counts. Given 500 edge insertions, *max* bar shows the longest time taken by an edge insertion, whereas *median* bar shows the median of the insertion times. *90%* bar shows the running time value such that 90 percentile of the edge insertions take at most that much time.

subgraphs are small, because they are bounded by higher K valued vertices, which in turn belong to their max-core. For large K values, subcores are bigger, because large K valued vertices tend to be close to each other due to the definition of k-core. Although their max-core sizes are small relative to that of small K valued vertices, their subcore sizes turn out to be larger.

5.6.5 Multihop Performance

In this set of experiments, we evaluate the impact of hop distance on the performance of multihop traversal insertion algorithm, given in Section 5.4.4, on real-world networks. We also evaluate the RCD maintenance algorithms. The goal is to observe how the maintenance times are effecting the total runtime for different hop distances and how the traversal space and time are reduced with increasing hop counts.

We show the normalized maintenance (white) and traversal (black) times for different hop counts (x-axis) in Figure 5.13. Normalization is done with respect to 2-hop results for each graph. We refer to the processing time taken by Lines 6 and 34 of Algorithm 20 as "maintenance" time and to the time taken by the rest of the algorithm as "traversal" time. For a given graph, we inserted the same set of 500 edges for each hop count, which are randomly selected at the start, so that the comparison is fair. Note that the bars corresponding to 2-hops represent the traversal based insertion algorithm, given in Section 5.4.3.

From the figure, we observe that the maintenance times are increasing with higher hop counts. For instance, 3-hop maintenance time is 86% more than the one for 2-hops, whereas the 4-hop and 5-hop times are 3.3× and 5.5× larger than the maintenance time used by the traversal algorithm (which uses 2-hop information). When we test the 6-hop traversal algorithm, it could be completed for 5 of the 9 graphs and gives 9.4 times slower running times, overall.

On the other hand, traversal times, shown with black bars, present a different picture. Higher number of hops result in reduced traversal spaces and thus lower running times. 3-hop traversal algorithm is up to 5× faster than the 2-hop variant. On average, 3-hop and 4-hop traversals are 2× and 3× faster than the 2-hop one. We observe that if the traversed graph space is high for the 2-hop algorithm (which is indicated by the long traversal times), it is likely to get better speedup with higher hop counts. For example, citationCiteseer graph requires large traversals when the 2-hop algorithm is applied. On average, the 2-hop traversal algorithm visits 55.3 vertices, and this number goes down to 12 and 2.8 with 3 and 4 hops, respectively. On the other hand, caidaRouterLevel graph requires 2.1 vertex traversals on average for the 2-hop algorithm and this number is going down to 1.5 with higher hop counts. Therefore, if the 2 hop algorithm traverses significant space, then there is room for improvement and this opportunity is leveraged well by the higher hop count algorithms. Otherwise, maintenance times become the bottleneck.

If we look at the total times, we see that 5 of the 9 graphs benefit from the higher hop counts and best performance is obtained by 3 or 4 hop algorithms for those graphs. One distinguishing feature of these 5 graphs is that, their 2-hop traversal times are more than 85% of the total time, which means that there is a significant room for improvement over the 2-hop variant.

To understand the running time changes with varying number of hops in a better way, we plotted the maximum, 90%, and median times for the real-world graphs in

Figure 5.14. We normalized all the times with respect to the 2-hop median times. Median times for all graphs are around 0.001 milliseconds, so they do not differ significantly for different hop counts. When we look at the 90 percentile bars, eu-2005 and power graphs show that 3-hop variant is superior to the 2-hop variant. Maximum times are ranging from 0.1 to 7 millisecond. The interesting thing about them is that all maximum time bars (except caidaRouterLevel and protein-interaction networks) show that 3-hop and 4-hop variants result in better running times compared to the 2-hop variants. This means that the larger hop counts reduce the variance in the edge insertion times, as they prevent very large traversals that the 2-hop variant sometimes encounters. These results confirm the fact that if there is a significant amount of work to do in order to adjust the k-core decomposition, then higher hops will provide better running times. Overall, we suggest to use 3-hops or 4-hops when the graph dataset used results in large traversals.

5.7 Related Work

The definition of k-core is first introduced by Seidman [161] to characterize the cohesive regions of graphs. Batagelj et al. [22] developed an efficient algorithm to find the k-core decomposition of a graph. In our work, we build upon these works to develop k-core decomposition algorithms that are incremental in nature, making it possible to apply these algorithms in streaming settings where edge insertions and removals happen frequently, such as maintaining a recent history of a dynamic graph.

There are many application areas of k-core decomposition including but not limited to social networks [71, 174], visualization of large networks [8, 186, 67], and protein interaction networks analysis [16, 182]. In social network analysis, k-cores has been used for community detection [71], clustering [174], and criminal network detection [132].

Thanks to its well-defined structure, k-cores has been used extensively to analyze the structure of certain types of networks [54, 113] and to generate graphs with specific properties [23]. Many graph problems like maximal clique finding [19], dense subgraph discovery [9], and betweenness approximation [82] use k-core decomposition as a subroutine.

In terms of algorithms specific to finding k-core decompositions, an external-memory algorithm for k-core decomposition is introduced in [42]. There are also studies about k-core decomposition on directed [70] and weighted [71] networks. As an effort to streaming k-core decomposition, Aksu et al. [6], introduced dense k-core subgraph maintenance algorithms in distributed settings. In their work, they materialized the k-core subgraph with large K values and maintain them for dynamic graphs. However, they ignore the maintenance of small K valued dense subgraphs. In this respect, our work is unique in the sense that we provide maintenance of *all* dense subgraphs in a given graph.

Concurrently with our work, Li et al. [107] published a report on incremental algorithms for core decomposition. Our algorithms differ from theirs in two important

aspects: (1) They propose quadratic complexity incremental algorithms, whereas our algorithms have linear complexity. (2) The speedup results achieved by our algorithm outperform theirs. For instance, their best algorithm has 6.3× speedup on the cond-mat graph, while our best algorithm (Traversal) achieves a speedup of 776.4×.

5.8 Summary

In this work we have introduced streaming algorithms for k-core decomposition of graphs. The key feature of these algorithms is their incremental nature — the ability to update the k-core decomposition quickly when a new edge is inserted or removed, without having to traverse the entire graph. Our experimental evaluation shows that these incremental algorithms can perform significantly better than their batch alternatives, where the speedup in execution time increases with the increasing graph size. Given the importance of k-core decomposition in detection of dense regions and communities, max. clique finding, and graph visualization, we believe these incremental algorithms will serve as a fundamental building block for future incremental solutions for other graph problems.

Algorithm 20: MULTIHOP TRAVERSAL:
INSERTEDGE($G(V,E), K(), RCD(,), n, u_1, u_2$)

Data: G: the graph, K: max-k values, RCD: residential core degrees, n: number of hops (> 1), (u_1,u_2): inserted edge

$r \leftarrow u_1$ ▷ Set the root
if $K(u_2) < K(u_1)$ **then** $r \leftarrow u_2$

$G \leftarrow G \cup (u_1, u_2)$ ▷ Add the edge into G
MULTIHOPPREPARERCDSINSERTION(G, K, RCD, n, u_1, u_2)
▷ Perform a traversal over vertices that have root's K value, while evicting the ones that cannot be a part of a k+1-core
$S \leftarrow$ empty stack ▷ To perform DFS
`visited`$[v] =$ **false**$, \forall v \in V$ ▷ To perform DFS (lazy init)
`evicted`$[v] =$ **false**$, \forall v \in V$ ▷ To remember evicted vert. (lazy init)
`cd`$[v] = 0, \forall v \in V$ ▷ To find vertices to be evicted (lazy init)
$k \leftarrow K(r)$ ▷ Remember the K value of the root
`cd`$[r] \leftarrow RCD(r, n)$ ▷ Set `cd` of root
S.push(r); `visited`$[r] \leftarrow$ **true**
while not S.empty() **do** ▷ Do a DFS traversal
    $v \leftarrow S$.pop()
    if `cd`$[v] > k$ **then** ▷ Vertex is currently part of a k+1-core
        for each $(v, w) \in E$ **do**
            ▷ Neighbouring vertex currently part of a k+1-core
            if $K(w) = k$ **and** $RCD(w, n-1) > k$ **and**
                not `visited`$[w]$ **then**
                S.push(w); `visited`$[w] \leftarrow$ **true**
                ▷ Use + as `cd`$[w]$ may be < 0 due to evictions
                `cd`$[w] \leftarrow$ `cd`$[w] + RCD(w, n)$
    else ▷ Vertex cannot be part of a k+1-core
        if not `evicted`$[v]$ **then** ▷ Recursively perform eviction
            PROPAGATEEVICTION($G, K,$ `cd`$,$ `evicted`$, k, v$)
`changed` $\leftarrow$ empty set ▷ For the vertices with updated K value
for each v s.t. `visited`$[v]$ **do** ▷ Find visited vertices
    if not `evicted`$[v]$ **then** ▷ If not evicted as well
        $K(v) \leftarrow K(v) + 1$ ▷ The vertex is part of a k+1-core
        `changed`.push(v);
MULTIHOPRECOMPUTERCDS($G, K, RCD, n,$ `changed`)

Algorithm 21: MULTIHOP RCD MAINTENANCE: MULTIHOPPREPARERCDSINSERTION $(G(V,E), K(), RCD(,), n, u_1, u_2)$

Data: G: the graph, K: max-k values, RCD: residential core degrees, n: number of hops (> 1), (u_1,u_2): inserted edge

```
1  r ← u1                                                          ▷ Set the root
2  if K(u2) < K(u1) then r ← u2
3  k ← K(r)                                          ▷ Remember the K value of the root
4  roots ← empty set
5  frontiers ← n number of empty sets
6  if K(u1) ≠ K(u2) then
7      for each h ∈ [1..n] do                                      ▷ For each hop
           ▷ r gets a new neig. with higher K value
8          RCD(r,h) ← RCD(r,h) + 1
9          if h < n and RCD(r,h) = k + 1 then
               ▷ If the RCD value exceeds k, it is pushed to frontiers. RCD of neigs will be updated in the next iteration
10             frontiers[h + 1].push(r)
11         if h > 1 then
               ▷ Neigs of the vertices in frontiers are explored to update their RCD values
12             for each v ∈ frontiers[h] do
13                 for each (v,w) ∈ E do
14                     if K(w) = k then
                           ▷ RCD value of every neig, with same K value, is incremented
15                         RCD(w,h) ← RCD(w,h) + 1
16                         if h < n and RCD(w,h) = k + 1 then
                               ▷ If the RCD value exceeds k, neigs will be updated in the next iteration
17                             frontiers[h + 1].push(w)
18 else
19     for each h ∈ [1..n] do                                      ▷ For each hop
20         if h = 1 then                                           ▷ MCD computation
               ▷ u1 and u2 get a new neig with equal K value
               ▷ If the RCD value exceeds k, it is pushed to frontiers. RCD of neigs will be updated in the next iteration
21             RCD(u1,h) ← RCD(u1,h) + 1
22             if RCD(u1,h) = k + 1 then
23                 frontiers[h + 1].push(u1)
24             RCD(u2,h) ← RCD(u2,h) + 1
25             if RCD(u2,h) = k + 1 then
26                 frontiers[h + 1].push(u2)
27         else
               ▷ Handle the newly inserted edge
28             if RCD(u2,h − 1) > k then
29                 RCD(u1,h) ← RCD(u1,h) + 1
30                 if h < n and RCD(u1,h) = k + 1 then
31                     frontiers[h + 1].push(u1)
32             if RCD(u1,h − 1) > k then
33                 RCD(u2,h) ← RCD(u2,h) + 1
34                 if h < n and RCD(u2,h) = k + 1 then
35                     frontiers[h + 1].push(u2)
               ▷ Neigs of the vertices in frontiers are explored to update their RCD values
36             for each v ∈ frontiers[h] do
37                 for each (v,w) ∈ E do
                       ▷ Exclude the newly inserted edge
38                     if not (v = u1 and w = u2) and
39                     not (v = u2 and w = u1) and
40                     K(w) = k then
41                         RCD(w,h) ← RCD(w,h) + 1
42                         if h < n and RCD(w,h) = k + 1 then
43                             frontiers[h + 1].push(w)
```

Algorithm 22: MULTIHOP RCD MAINTENANCE: MULTIHOPPREPARERCDSREMOVAL $(G(V,E), K(), RCD(,), n, u_1, u_2)$

Data: G: the graph, K: max-k values, RCD: residential core degrees, n: number of hops (> 1), (u_1,u_2): inserted edge

```
1  r ← u1                                                          ▷ Set the root
2  if K(u2) < K(u1) then r ← u2
3  k ← K(r)                                                        ▷ Remember the K value of the root
4  roots ← empty set
5  frontiers ← n number of empty sets
6  if K(u1) ≠ K(u2) then
7      for each h ∈ [1..n] do                                      ▷ For each hop
8          RCD(r,h) ← RCD(r,h) − 1
9          if h < n and RCD(r,h) = k then
10             frontiers[h+1].push(r)
11         if h > 1 then
12             for each v ∈ frontiers[h] do
13                 for each (v,w) ∈ E do
14                     if K(w) = k then
15                         RCD(w,h) ← RCD(w,h) − 1
16                         if h < n and RCD(w,h) = k then
17                             frontiers[h+1].push(w)
18 else
       ▷ Remember the old RCD values of u1 and u2
19     old_RCD ← empty set for u1 and u2
20     for each h ∈ [1..n] do                                      ▷ For each hop
21         old_RCD(u1,h) ← RCD(u1,h)
22         old_RCD(u2,h) ← RCD(u2,h)
23     for each h ∈ [1..n] do                                      ▷ For each hop
24         if h = 1 then
25             RCD(u1,h) ← RCD(u1,h) − 1
26             if RCD(u1,h) = k then
27                 frontiers[h+1].push(u1)
28             RCD(u2,h) ← RCD(u2,h) − 1
29             if RCD(u2,h) = k then
30                 frontiers[h+1].push(u2)
31         else
32             if old_RCD(u2,h−1) > k then
33                 RCD(u1,h) ← RCD(u1,h) − 1
34                 if h < n and RCD(u1,h) = k then
35                     frontiers[h+1].push(u1)
36             if old_RCD(u1,h−1) > k then
37                 RCD(u2,h) ← RCD(u2,h) − 1
38                 if h < n and RCD(u2,h) = k then
39                     frontiers[h+1].push(u2)
40             for each v ∈ frontiers[h] do
41                 for each (v,w) ∈ E do
42                     if not (v = u1 and w = u2) and
43                     not (v = u2 and w = u1) and
44                     K(w) = k then
45                         RCD(w,h) ← RCD(w,h) − 1
46                         if h < n and RCD(w,h) = k then
47                             frontiers[h+1].push(w)
```

Algorithm 23: MULTIHOP RCD MAINTENANCE:
MULTIHOPRECOMPUTERCDS $(G(V,E), K(), RCD(,), n, \texttt{changed})$

Data: G: the graph, K: max-k values, RCD: residential core degrees, n: number of hops (> 1), `changed`: set of vertices with updated K value

```
1  visited[v] = false, ∀v ∈ V                                   ▷ Lazy init
2  for each v ∈ changed do
3      visited[v] = true
4  for each h ∈ {1...n} do
5      updated ← empty set
6      for each v ∈ changed do
7          for each (v, w) ∈ E do
8              if not visited[w] and
               ▷ For insertion
9                                  (K(w) = K(v) or K(w) = K(v) − 1) then
                   ▷ For removal
                   ▷                          (K(w) = K(v) or K(w) = K(v) + 1)
10                 updated.push(w)
11                 visited[w] = true
12     for each v ∈ updated do
13         changed.push(v)
14     for each v ∈ changed do
15         RCD(h)(v) ← COMPUTERCD (v, K, RCD, h)
```

Graph scale	with RCD (ratio)	without RCD
16	0.032 (%48)	0.067
18	0.175 (%52)	0.335
20	1.041 (%50)	2.047
22	4.218 (%49)	8.600
24	6.098 (%67)	8.991

Table 5.2: Average runtimes (secs) for one edge removal plus one edge insertion with traversal algorithm on Erdös-Renyi graphs. Ratio shows with RCD runtimes relative to without.

Chapter 6: Finding the Hierarchy of Dense Subgraphs using Nucleus Decompositions

Graphs are widely used to model relationships in a wide variety of domains such as sociology, bioinformatics, infrastructure, the WWW, to name a few. One of the key observations is that while real-world graphs are often globally sparse, they are locally dense. In other words, the average degree is often quite small (say at most 10 in a million vertex graph), but vertex neighborhoods are often dense. The classic notions of transitivity [179] and clustering coefficients [180] measure these densities, and are high for many real-world graphs [143, 163].

Finding dense subgraphs is a critical aspect of graph mining [105]. It has been used for finding communities and spam link farms in web graphs [101, 72, 56], graph visualization [7], real-time story identification [11], DNA motif detection in biological networks [64], finding correlated genes [185], epilepsy prediction [88], finding price value motifs in financial data [57], graph compression [34], distance query indexing [91], and increasing the throughput of social networking site servers [73]. This is closely related to the classic sociological notion of group cohesion [24, 61]. There are

tangential connections to classic community detection, but the objectives are significantly different. Community definitions involve some relation of inner versus outer connections, while dense subgraphs purely focus on internal cohesion.

6.1 Introduction

Our input is a graph $G = (V, E)$. For vertex set S, we use $E(S)$ to denote the set of edges internal to S. The *edge density* of S is $\rho(S) = |E(S)|/\binom{|S|}{2}$, the fraction of edges in S with respect to the total possible. The aim is to find a set S with high density subject to some size constraint. Typically, we are looking for large sets of high density.

In general, one can define numerous formulations that capture the main problem. The maximum clique problem is finding the largest S where $\rho(S) = 1$. Finding the densest S of size at least k is the k-densest subgraph problem. Quasi-cliques, as defined recently by Tsourakakis et al. [170], are sets that are almost cliques, up to some fixed "defect." Unfortunately, most formulations of finding dense subgraphs are NP-hard, even to approximate [86, 60, 95].

For graph analysis, one rarely looks for just a single (or the optimal, for whatever notion) dense subgraph. We want to find many dense subgraphs and understand the relationships among them. Ideally, we would like to see if they nest within each other, if the dense subgraphs are concentrated in some region, and if they occur at various scales of size and density. Our work is motivated by the following questions.

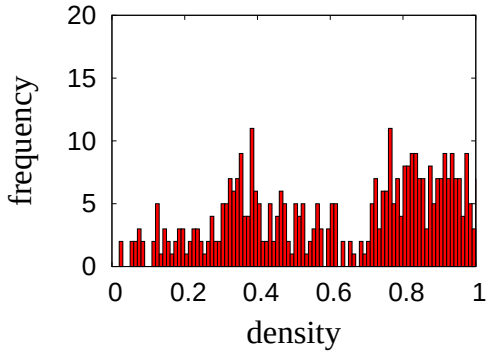


Figure 6.1: Density histogram of `facebook` $(3,4)$-nuclei. 145 nuclei have density of at least 0.8 and 359 nuclei are with the density of more than 0.25.

- How do we attain a global, hierarchical representation of many dense subgraphs in a real-world graph?
- Can we define an efficiently solvable objective that directly provides *many* dense subgraphs? We wish to avoid heuristics, as they can be difficult to predict formally.

6.1.1 Our contributions

Nucleus decompositions: Our primary theoretical contribution is the notion of *nuclei* in a graph. Roughly speaking, an (r, s)-nucleus, for fixed (small) positive integers $r < s$, is a maximal subgraph where every r-clique is part of many s-cliques. (The real definition is more technical and involves some connectivity properties.) Moreover, nuclei that do not contain one another cannot share an r-clique. This is

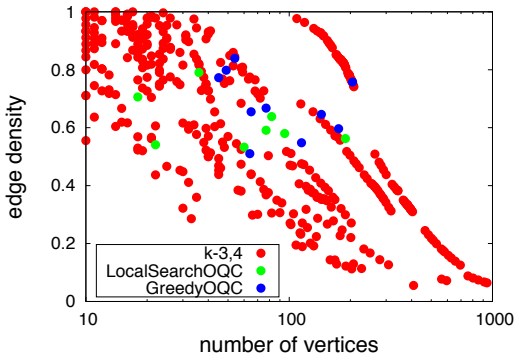


Figure 6.2: Size vs. density plot for `facebook` $(3,4)$-nuclei. 50 nuclei are larger than 30 vertices with the density of at least 0.8. There are also 138 nuclei larger than 100 vertices with density of at last 0.25.

inspired by and is a generalization of the classic notion of k-cores, and also k-trusses (or triangle cores).

We show that the (r,s)-nuclei (for any $r < s$) form a hierarchical decomposition of a graph. The nuclei are progressively denser as we go towards the leaves in the decomposition. We provide an exact, efficient algorithm that finds all the nuclei and builds the hierarchical decomposition. In practice, we observe that $(3,4)$-nuclei provide the most interesting decomposition. We find the $(3,4)$-nuclei for a large variety of more than 20 graphs. Our algorithm is feasible in practice, and we are able to process a 39 million edge graph in less than an hour (using commodity hardware). The source code of our algorithms are available [5].

[5]http://bmi.osu.edu/hpc/software/nucleus

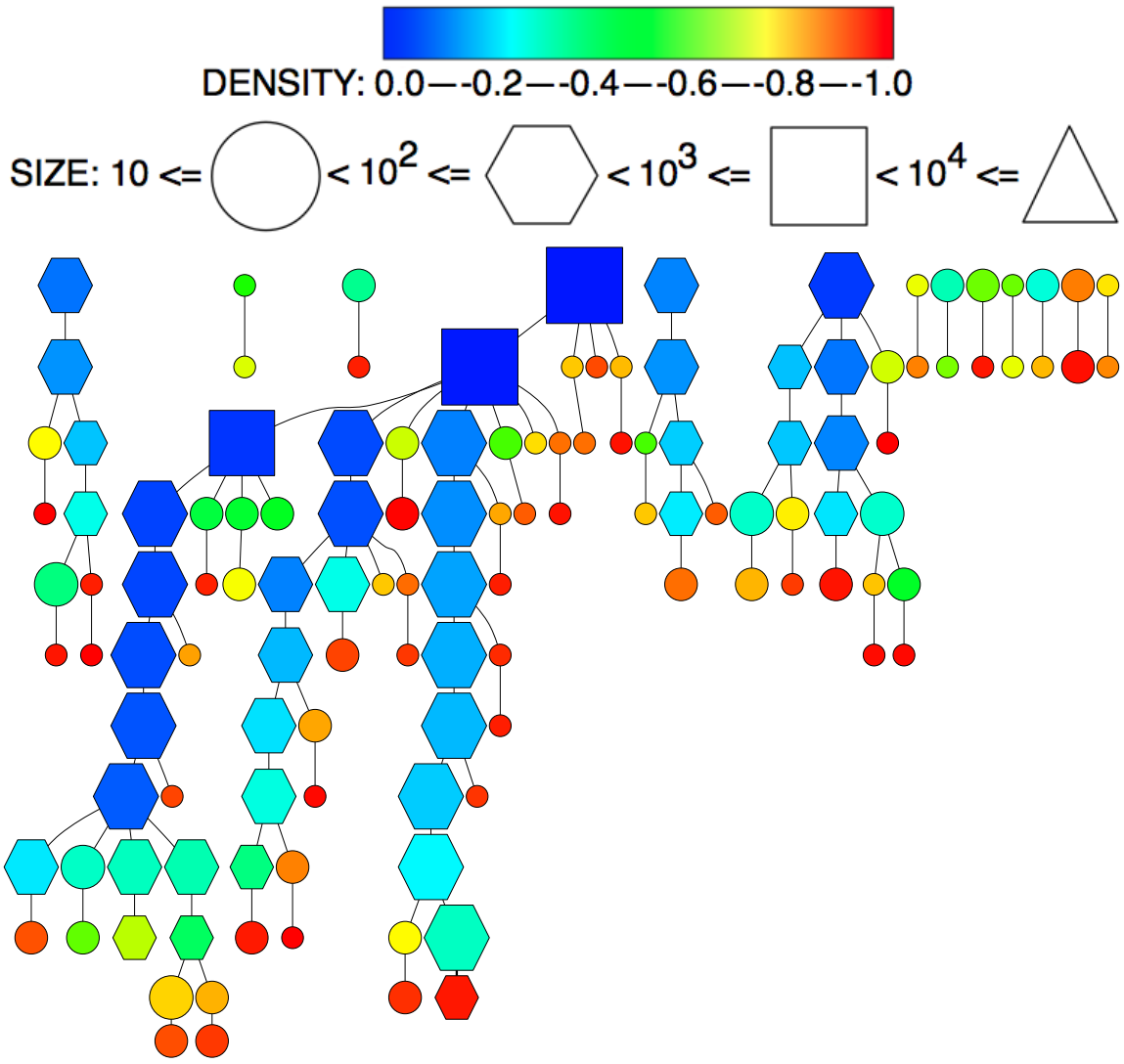


Figure 6.3: $(3,4)$-nuclei forest for `facebook`. Legends for densities and sizes are shown at the top. Long chain paths are contracted to single edges. In the uncontracted forest, there are 47 leaves and 403 nuclei. Branching depicts the different regions in the graph, 13 connected components exist in the top level. Sibling nuclei have limited overlaps up to 7 vertices.

Dense subgraphs from $(3,4)$**-nuclei:** The $(3,4)$-nuclei provide a large set of dense subgraphs for range of densities and sizes. For example, there are 403 $(3,4)$-nuclei (of size at least 10 vertices) in a `facebook` network of 88K edges. We show the density histogram of these nuclei in Fig. 6.1, plotting the number of nuclei with a given density. Observe that we get numerous dense subgraphs, and many with density fairly close to 1. In Fig. 6.2, we present a scatter plot of vertex size vs density of the $(3,4)$-nuclei. Observe that we obtain dense subgraphs over a wide range of sizes. For comparison, we also plot the output of recent dense subgraph algorithms from Tsourakakis et al. [170]. (These are arguably the state-of-the-art. More details in next section.) Observe that $(3,4)$-nuclei give dense subgraphs of comparable quality. In some cases, the output of [170] is very close to a $(3,4)$-nucleus.

Representing a graph as forest of $(3,4)$**-nuclei:** We build the forest of $(3,4)$-nuclei for all graphs experimented on. An example output is that of Fig. 6.3, the forest of $(3,4)$-nuclei for the `facebook` network. Each node of the forest is a $(3,4)$-nucleus, and tree edges indicate containment. More generally, an ancestor nucleus contains all descendant nuclei. By the properties of $(3,4)$-nuclei, any two incomparable nodes do not share a triangle. So the branching in the forest represents different regions of the graph. (All nuclei of less than 10 vertices are omitted. For presentation, we contract long chain paths in the tree to single edges, so the forest has less than 403 nodes.)

In the nuclei figures, densities are color-coded, with hotter colors indicating higher density. The log of sizes are coded by shape (circles comprise between 10 and 100

vertices, hexagons between 100 and 1000 vertices, etc.) For a fixed shape, relative size corresponds to relative size in number of vertices. We immediately see the hierarchy of dense structures. Observe the colors becoming hotter as we go towards to leaves, which are mostly red (density > 0.8). We see numerous hexagons and large circles of color between light blue to green. These indicate the larger parent subgraphs of moderate density (actually density of say 0.25 is fairly high for a subgraph having many hundreds of vertices).

The branching is also significant, and we can group together the dense subgraphs according to the hierarchy. We observe such branching in all our experiments, and show more such results later in the chapter. The $(3, 4)$-nuclei provide a simple, hierarchical visualization of dense substructures. They are well-defined and their exact computation is algorithmically feasible and practical.

We also want to emphasize the overlap between sibling nuclei. While sibling nuclei cannot share triangles, they can share edges, thus vertices. We observe roughly 20 pairs of $(3, 4)$-nuclei having intersections of 4-6 vertices. For larger graphs, we observe many more pairs of intersecting nuclei (with larger intersections).

The rest of the chapter is organized as follows: §6.2 summarizes the related work, §6.3 introduces the main definitions and the lemma about the nucleus decomposition, §6.4 gives the algorithm to generate a nucleus decomposition and provides a complexity analysis, §6.5 contains the results of extensive experiments we have, and §6.6 concludes the chapter by discussing the future directions.

6.2 Previous work

Dense subgraph algorithms: As discussed earlier, most formulations of the densest subgraph problem are NP-hard. Some variants such as maximum average degree [75, 68] and the recently defined triangle-densest subgraph [171] are polynomial time solvable. Linear time approximation algorithms have been provided by Asashiro et al. [13], Charikar [40], and Tsourakakis [171]. There are numerous recent practical algorithms for various such objectives: Andersen and Chellapilla's use of cores for dense subgraphs [10], Rossi et al.'s heuristic for clique [140], Tsourakakis et al.'s notion of quasi-cliques [170]. These algorithms are extremely efficient and produce excellent output. For comparison's sake, we consider Tsourakakis et al. [170] as the state-of-the-art, which was compared with previous core-based heuristics and is much superior to prior art. Indeed, their algorithms are elegant, extremely efficient, and provide high quality output (and much faster than ours. More discussion in §7.7.2). These methods are tailored to finding one (or a few) dense subgraphs, and do not give a global/hierarchical view of the structure of dense subgraphs. We believe it would be worthwhile to relate their methods with our notion of nuclei, to design even better algorithms.

k-cores and k-trusses: The concepts of k-cores and k-trusses form the inspiration for our work. A k-core is a maximal subgraph where each vertex has minimum degree k, while a k-truss is a subgraph where each edge participates in at least k triangles. The first definition of k-cores was given by Erdős and Hajnal [58]. It has

been rediscovered numerous times in the context of graph orientations and is alternately called the coloring number and degeneracy [109, 161]. The first linear time algorithm for computing k-cores was given by Matula and Beck [118]. The earliest applications of cores to social networks was given by Seidman [161], and it is now a standard tool in the analysis of massive networks. The notions of k-truss or triangle-cores were independently proposed by Cohen [45], Zhang and Parthasarathy [187], and Zhao and Tung [189] for finding clusters and for network visualization. They all provide efficient algorithms for these decompositions, and Cohen [45] and Wang and Cheng [177] explicitly focus on massive scale. In [178], Wang et al. proposed DN-graph, a similar concept to k-truss, where each edge should be involved in k triangles, and adding or removing a vertex from DN-graph breaks this constraint. Apart from the k-core and k-truss definitions, k-plex and k-club subgraph definitions have drawn a lot of interest as well. In a k-plex subgraph, each vertex is connected to all but at most $k-1$ other vertices [162], which complements the k-core definition. In a k-club subgraph, the shortest path from any vertex to other vertex is not more than k [125]. All these methods find subgraphs of moderate density, and give a global decomposition to visualize a graph.

Simplicial complexes: Our nucleus decomposition definition has some connections to simplicial complexes in algebraic topology. Algebraic topology is the subject of spaces and maps between them using algebraic methods [126]. Simplicial complex is a topological space of a certain kind of structures, such as vertices, edges, triangles,

etc. Barcelo et al. [21] gives a detailed analysis related to connectivity properties of simplicial complexes and introduces similar definitions to the our $\mathcal{S}$-*connectedness* definition. Betti number of a simplicial complex [127] is another related concept, which are used to distinguish topological spaces based on the connectivity of n-dimensional simplicial complexes. There are some studies on the connectivity properties of simplicial complexes in more general spaces as well, and more information can be found in [126].

6.3 Nucleus decomposition

Our main theoretical contribution is the notion of nucleus decompositions. We have an undirected, simple graph G. We use K_r to denote an r-clique and start with some technical definitions.

Definition 10. *Let $r < s$ be positive integers and $\mathcal{S}$ be a set of K_ss in G.*

- *$K_r(\mathcal{S})$ the set of K_rs contained in some $S \in \mathcal{S}$.*
- *The number of $S \in \mathcal{S}$ containing $R \in K_r(\mathcal{S})$ is the* $\mathcal{S}$-degree *of that K_r.*
- *Two K_rs R, R' are* $\mathcal{S}$-connected *if there exists a sequence $R = R_1, R_2, \ldots, R_k = R'$ in $K_r(\mathcal{S})$ such that for each i, some $S \in \mathcal{S}$ contains $R_i \cup R_{i+1}$.*

These definitions are generalizations of the standard notion of the vertex degree and connectedness. Indeed, setting $r = 1$ and $s = 2$ (so $\mathcal{S}$ is a set of edges) yields exactly that. Our main definition is as follows.

Definition 11. *Let k, r, and s be positive integers such that $r < s$. A k-(r, s)*-nucleus *is a maximal union $\mathcal{S}$ of K_ss such that:*

- *The $\mathcal{S}$-degree of any $R \in K_r(\mathcal{S})$ is at least k.*
- *Any $R, R' \in K_r(\mathcal{S})$ are $\mathcal{S}$-connected.*

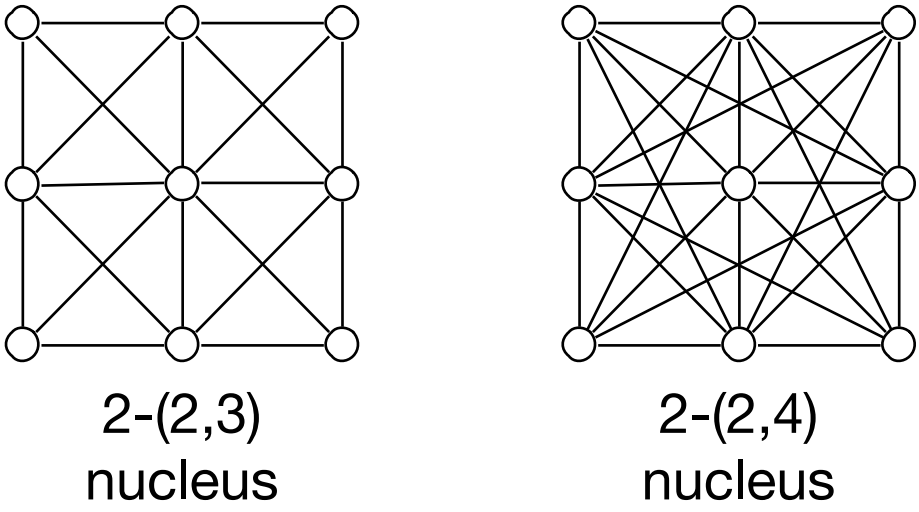


Figure 6.4: Having same number of vertices, 2-$(2,4)$ nucleus is denser than 2-$(2,3)$.

We simply refer to (r,s)-nuclei when k is unspecified. Note that we treat nuclei as a union of cliques, though eventually, we look at this as a subgraph. Our theoretical treatment is more convenient in the former setting, and hence we stick with this definition. In our applications, we simply look at nuclei as subgraphs.

Intuitively, a nucleus is a tightly connected cluster of cliques. For large k, we expect the cliques in $\mathcal{S}$ to intersect heavily, creating a dense subgraph. For a fixed k, r and same number of vertices, the density of the nuclei increases, as we increase s. Consider the example of Fig. 6.4, where there is a 2-$(2,3)$-nucleus and a 2-$(2,4)$-nucleus on the same number of vertices. Since in the latter case, we need every edge to participate in at least 2 K_4s, the resulting density is much higher.

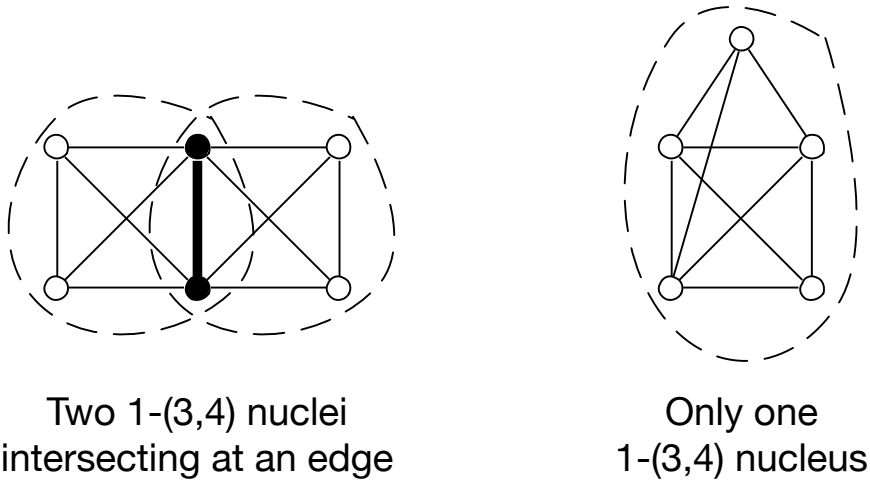


Figure 6.5: The left figure shows two $(3, 4)$-nuclei overlapping at an edge. The right figure has only one $(3, 4)$-nucleus

As stated earlier, our definitions are inspired by k-cores and k-trusses. Set $r = 1, s = 2$. A k-$(1, 2)$-nucleus is a maximal (induced) connected subgraph with minimum vertex degree k. *This is exactly a k-core.* Setting $r = 2, s = 3$ gives maximal subgraphs where every edge participates in at least k triangles, and edges are triangle-connected. This is essentially the definition of k-trusses or triangle-cores.

So far we only discussed the degree constraint of nuclei. Note that a nucleus is not just connected in the usual (edge) sense, but requires the stronger property of being $\mathcal{S}$-connected. The standard definitions of trusses or triangle-cores omit the triangle-connectedness. For us, this is critical. Two cliques of distinct (r, s)-nuclei *can* intersect. For example, when $r > 2$, nuclei can have edge overlaps. This allows for finding even denser subgraphs, as Fig. 6.5 shows. In the left, cores, trusses, etc.

pick up the entire graph. But there are actually 2 different 1-$(3,4)$-nuclei (each K_4) intersecting at an edge. The $(3,4)$-nuclei are denser than the graph itself. Note that any edge disjoint decomposition would not find two dense subgraphs.

Critically, the set of (r,s)-nuclei form a laminar family. A laminar family is a set system where all pairwise intersections are trivial (either empty or contains one of the sets).

Lemma 3. *The family of (r,s)-nuclei form a laminar family.*

Proof. Consider k-(r,s)-nucleus $\mathcal{S}$ and k'-(r,s)-nucleus $\mathcal{S}'$, where $k \leq k'$. Suppose they had a non-empty intersection, so some $K_s(S)$ is contained in both $\mathcal{S}$ and $\mathcal{S}'$. Observe that K_rs in $K_r(\mathcal{S})$ are connected to K_rs in $K_r(\mathcal{S}')$. Furthermore, the $(\mathcal{S} \cup \mathcal{S}')$-degree of member of $K_r(\mathcal{S} \cup \mathcal{S}')$ is at least k. Hence $\mathcal{S} \cup \mathcal{S}'$ satisfies the two conditions of being a nucleus, except maximality. By $\mathcal{S}$ is a k-(r,s)-nucleus, so $\mathcal{S} \cup \mathcal{S}' = \mathcal{S}$. So any non-empty intersection is trivial. □

Consider two nuclei that are not ancestor-descendant. By the above lemma, these two nuclei (considered as subgraphs of G) cannot share a K_s. Actually, the argument above proves that they cannot even share a K_r. This is the key disjointness property of nuclei.

Every laminar family is basically a hierarchical set system. Alternately, every laminar family can be represented by a forest of containment. For every nucleus $\mathcal{S}$, any other nucleus intersecting $\mathcal{S}$ is either contained in $\mathcal{S}$ or contains $\mathcal{S}$. Furthermore, all these sets are nested in each other. It makes sense to talk of the smallest sized nucleus containing $\mathcal{S}$. This leads to the main construct we use to represent nuclei.

Definition 12. *Fix $r < s$. Define the* forest of (r,s)-nuclei *as follows. There is a node for each (r,s) nucleus. The parent of every nucleus is the smallest (by cardinality) other nucleus containing it.*

In our figures, we will only show the internal nodes of out degree at least 2, and contract any path of out degree 1 vertices into a single path. This preserves all the branching of the forest.

6.4 Generating nucleus decompositions

Our primary algorithmic goal is to construct the tree of nuclei. The algorithm is a direct adaptation of the classic Matula-Beck result of getting k-cores in linear time [118]. There are numerous technicalities involved in generalizing the proof. Intuitively, we do the following. Construct a graph $\mathcal{H}$ where the nodes are all K_rs of G and there is an edge connecting two K_rs if they are contained in a single K_s of G. We then perform a core decomposition on $\mathcal{H}$. Actually, this does not work. Edges of G (obviously) contain exactly 2 vertices of G, and the procedure above exactly produces nuclei for $r = 1, s = 2$. In general, a K_s contains $\binom{s}{r}$ K_rs, and the graph analogy above is incorrect. At some level, we are performing a hypergraph version of Matula-Beck. The proofs therefore need to be adapted to this setting.

Analogous to k-cores, the main procedure `set-k` (Algorithm 24) assigns a number, denoted by $\kappa(\cdot)$, to each K_r in G.

It is convenient to denote the set of K_rs in G by $R_1, R_2, \ldots$, where R_i is the ith processed K_r in `set-k`. We will refer to this index as *time*. When we say "at time t", we mean at the beginning of the iteration where R_t is processed.

Claim 1. *The sequence $\{\kappa(R_i)\}$ is monotonically non-decreasing.*

Proof. This holds because the loop goes R in non-decreasing order of $d(R)$ and Step 11 ensures that no new value of $\delta(\cdot)$ decreases below the current $\kappa(R)$. □

Algorithm 24: `set-k`(G, r, s)

```
Enumerate all K_r s and K_s s in G(V, E)
For every K_r R, initialize δ(R) to be the number of K_s s containing R
Mark every K_r as unprocessed
for each unprocessed K_r R with minimum δ(R) do
    κ(R) = δ(R)
    Find set S of K_s s containing R
    for each S ∈ S do
        if any K_r R' ⊂ S is processed then
            Continue
        for each K_r R' ⊂ S, R' ≠ R do
            if δ(R') > δ(R) then
                δ(R') = δ(R') − 1
    Mark R as processed
return array κ(·)
```

- Because of Claim 1, we can define *transition time* t_i to be the first time when the κ-value becomes i. Formally, t_i is the unique index such that $\kappa(R_{t_i}) = i$ and $\kappa(R_{t_i-1}) < i$.

- We say K_s S is *unprocessed at time* t if all $R \in K_r(S)$ are unprocessed at time t. This set of K_ss is denoted by $\mathcal{S}_t$.

- The *supergraph* $\mathcal{G}_t$ has node set $K_r(\mathcal{S}_t)$, and $R, R' \in K_r(\mathcal{S}_t)$ are connected by a link if $R \cup R'$ is contained in some K_s of $\mathcal{S}_t$. Links are associated with elements of $\mathcal{S}_t$ (and there may be multiple links between R and R').

We prove an auxiliary claim relating the $\delta(\cdot)$ values to $\mathcal{S}_t$.

Claim 2. *At time t, for any unprocessed K_r R, $\delta(R)$ is at least the $\mathcal{S}_t$-degree of R. If $t = t_k$ (for some k), then $\delta(R)$ is exactly the $\mathcal{S}_t$-degree of R.*

Proof. Pick unprocessed R'. The value of $\delta(R)$ is initially the number of K_ss containing R'. It is decremented only in Step 12, which happens only when a processed K_s containing R' is found. (Sometimes, the decrement will still not happen, because of Step 11.) Hence, the value of $\delta(R')$ at time t is at least the number of unprocessed K_ss containing R'.

Suppose $t = t_k$. For any preceding $\hat{t} < t$, the current $\kappa(\cdot)$ value is always at most k. For unprocessed (at time t) R, $\delta(R) > k$. Hence the decrement of Step 12 will always happen, and $\delta(R)$ is exactly the $\mathcal{S}_t$-degree of R. □

Claim 3. *Every k-(r,s)-nucleus is contained in $\mathcal{S}_{t_k}$.*

Proof. Consider k-(r,s)-nucleus $\mathcal{S}$. Take the first $R \in K_r(\mathcal{S})$ that is processed. At this time (say t), no $S \in \mathcal{S}$ can be processed. Hence, $\mathcal{S} \subseteq \mathcal{S}_t$. By Claim 2, $d(R)$ is at least the $\mathcal{S}_t$-degree of R, which is at least the $\mathcal{S}$-degree of R. The latter is at least k, since $\mathcal{S}$ is a k-(r,s)-nucleus. By definition of t_k, $t \geq t_k$ and hence $\mathcal{S}_t \subseteq \mathcal{S}_{t_k}$. Thus, $\mathcal{S} \subseteq \mathcal{S}_{t_k}$. □

The main lemma shows that the output of `set-k` essentially tells us the nuclei.

Lemma 4. *The k-(r,s)-nuclei are exactly the links (which are K_ss) of connected components of $\mathcal{G}_{t_k}$.*

Proof. Consider k-(r,s)-nucleus $\mathcal{S}$. By Claim 3, it is contained in $\mathcal{S}_{t_k}$. By the nucleus definition, $\mathcal{S}$ is connected (as links) in $\mathcal{G}_{t_k}$. Let $\mathcal{S}'$ be the (set of links) connected component of $\mathcal{G}_{t_k}$ containing $\mathcal{S}$. By Claim 2, at time t_k, for any $R \in K_r(\mathcal{S}')$, $\delta(R)$ is exactly the $\mathcal{S}_{t_k}$-degree of R. Since $\mathcal{S}'$ is a connected component of $\mathcal{G}_{t_k}$, the $\mathcal{S}_{t_k}$-degree is the $\mathcal{S}'$-degree, which in turn is at least k. In other words, $\mathcal{S}'$ satisfies both conditions of being a k-(r,s)-nucleus, except maximality. By maximality of $\mathcal{S}$, $\mathcal{S} = \mathcal{S}'$. □

Building the forest of nuclei: From Lem. 4, it is fairly straightforward to get all the nuclei. First run `set-k` to get the processing times and the $\kappa(\cdot)$ values. We can then get all t_k times as well. Suppose for any K_r in G, we can access all the K_ss containing it. Then, it is routine to traverse $\mathcal{G}_{t_k}$ to get the links of connected components. To avoid traversing the same component repeatedly, we produce nuclei in reverse order of k. In other words, suppose all connected components of $\mathcal{G}_{t_{k+1}}$

have been determined. For $\mathcal{G}_{t_k}$, it suffices to determine the connected components involving nodes processed in time $[t_k, t_{k+1})$. Any time a traversal encounters a node in $\mathcal{G}_{t_{k+1}}$, we need not traverse further. This is because all other connected nodes of $\mathcal{G}_{t_{k+1}}$ are already known from previous traversals. We do not get into the data structure details here, but it suffices to visit all nodes and links of $\mathcal{G}_0$ exactly once.

6.4.1 Bounding the complexity

There are two options of implementing this algorithm. The first is faster, but has forbiddingly large space. The latter is slower, but uses less space. In practice, we implement the latter algorithm. We use $ct_r(v)$ for the number of K_rs containing v and $ct_r(G)$ for the total number of K_rs in G. We denote by $RT_r(G)$ the running time of an arbitrary procedure that enumerates all K_rs in G.

Theorem 10. *It is possible to build the forest of nuclei in $O(RT_r(G) + RT_s(G))$ time with $O(ct_r(G) + ct_s(G))$ space.*

Proof. This is the obvious implementation. The very first step of `set-k` requires the clique enumeration. Suppose we store the global supergraph $\mathcal{G} = \mathcal{G}_0$. This has a node for every K_r in G and a link for every K_s in G. The storage is $O(ct_r(S) + ct_s(G))$. From this point onwards, all remaining operations are linear in the storage. This is by the analysis of the standard core decomposition algorithm of Matula and Beck [118]. Every time we process a K_r, we can delete it and all incident links from $\mathcal{G}$. Every link is touched at most a constant number of times during the entire running on `set-k`. As explained earlier, we can get all the nuclei by a single traversal of $\mathcal{G}$. □

Theorem 11. *It is possible to build the forest of nuclei in $O(RT_r(G)+\sum_v ct_r(v)d(v)^{s-r})$ time with $O(ct_r(G))$ space.*

Proof. Instead of explicitly building $\mathcal{G}$, we only build adjacency lists when required. The storage is now only $O(ct_r(G))$. In other words, given a K_r R, we find all K_ss containing R only when R is processed/traversed. Each R is processed or traversed at most once in `set-k` and the forest building. Suppose R has vertices $v_1, v_2, \ldots, v_r$.

We can find all K_ss containing R by looking at all $(s-r)$-tuples in each of the neighborhoods of v_i. (Indeed, it suffices to look at just one such neighborhood.) This takes time at most $\sum_R \sum_{v \in R} d(v)^{s-r} = \sum_v \sum_{R \ni v} d(v)^{s-r} = \sum_v ct_r(v) d(v)^{s-r}$. □

Let us understand these running times. When $r < s \leq 3$, it clearly benefits to go with Thm. 10. Triangle enumeration is a well-studied problem and there exist numerous optimized, parallel solutions for the problem. In general, the classic triangle enumeration of Chiba and Nishizeki takes $O(m^{3/2})$ [44] and is much better in practice [46, 160, 169]. This completely bounds the time and space complexities.

For our best results, we build the $(3,4)$-nuclei, and the number of K_4s is too large to store. We go with Thm. 11. The storage is now at most the number of triangles, which is manageable. The running time is basically bounded by $O(\sum_v ct_r(v)d(v))$. The number of triangles incident to v, $ct_3(v)$ is $cc(v)d(v)^2$, where $cc(v)$ is the clustering coefficient of v. We therefore get a running time of $O(\sum_v cc(v)d(v)^3)$. This is significantly superlinear, but clustering coefficients generally decay with degree [143, 163]. Overall, the implementation can be made to scale to tens of millions of edges with little difficulty.

6.5 Experimental Results

We applied our algorithms to large variety of graphs, obtained from SNAP [166] and UF Sparse Matrix Collection[1]. The vital statistics of these graphs are given in Tab. 7.1. All the algorithms in our framework are implemented in C++ and compiled with `gcc 4.8.1` at -O2 optimization level. All experiments are performed on a Linux

	$\|V\|$	$\|E\|$	Description	$\sum_v c_3(v)d(v)$	$(3,4)$ time (sec)	[170] Density (size)	(3,4)-nucleus Density (size)
dolphins	62	159	Biological	2.2K	< 1	0.68(8)	0.71(8)
polbooks	105	441	US Politics Books	23.8K	< 1	0.67(13)	0.62(13)
adjnoun	112	425	Adj. and Nouns	17.6K	< 1	0.60(15)	0.22(32)
football	115	613	World Soccer 98	26.3K	< 1	0.89(10)	0.89(10)
jazz	198	$2.74K$	Musicians	2.3M	< 1	1.00(30)	1.00(30)
celegans n.	297	$2.34K$	Biological	418K	< 1	0.61(21)	0.91(10)
celegans m.	453	$2.04K$	Biological	565K	< 1	0.67(17)	0.64(18)
email	$1.13K$	$5.45K$	Email	1.2M	< 1	1.00(12)	1.00(12)
facebook	$4.03K$	$88.23K$	Friendship	712M	93	0.83(54)	0.98(109)
protein_inter.	$9.67K$	$37.08K$	Protein Inter.	35M	< 1	1.00(11)	1.00(11)
as-22july06	$22.96K$	$48.43K$	Autonomous Sys.	199M	< 1	0.58(12)	1.00(18)
twitter	$81.30K$	$2.68M$	Follower-Followee	1.8B	396	0.85(83)	1.00(26)
soc-sign-epinions	$131.82K$	$841.37K$	Who-trust-whom	1.4B	242	0.71(79)	1.00(112)
coAuthorsCiteseer	$227.32K$	$814.13K$	CoAuthorship	2.1B	50.1	1.00(87)	1.00(87)
citationCiteseer	$268.49K$	$1.15M$	Citation	297M	3.4	0.71(10)	1.00(13)
web-NotreDame	$325.72K$	$1.49M$	Web	33.9B	671	1.00(151)	1.00(155)
amazon0601	$403.39K$	$3.38M$	CoPurchase	802M	23	1.00(11)	1.00(11)
web-Google	$875.71K$	$5.10M$	Web	11.4B	163	1.00(46)	1.00(33)
com-youtube	$1.13M$	$2.98M$	Social	451M	43	0.49(119)	0.92(24)
as-skitter	$1.69M$	$11.09M$	Autonomous Sys.	1.6B	1,036	0.53(319)	0.94(91)
wikipedia-2005	$1.63M$	$19.75M$	Wikipedia Link	741B	1,312	0.53(33)	0.82(14)
wiki-Talk	$2.39M$	$5.02M$	Wikipedia User	136B	605	0.48(321)	0.59(95)
wikipedia-200609	$2.98M$	$37.26M$	Wikipedia Link	2,015B	2,830	0.49(376)	0.62(103)
wikipedia-200611	$3.14M$	$39.38M$	Wikipedia Link	2,197B	3,039	1.00(55)	1.00(32)

Table 6.1: Important statistics for the real-world graphs of different types and sizes. Largest graph in the dataset has more than $39M$ edges. Times are in seconds. Density of subgraph S is $|E(S)|/\binom{|S|}{2}$ where $E(S)$ is the set of edges internal to S. Sizes are in number of vertices.

operating system running on a machine with two Intel Xeon E5520 2.27 GHz CPUs, with 48GB of RAM.

We computed the (r, s)-nuclei for all choices of $r < s \leq 4$, but do not present all results for space considerations. We mostly observe that the forest of $(3, 4)$-nuclei provides the highest quality output, both in terms of hierarchy and density.

As mentioned earlier, we will now treat the nuclei as just induced subgraphs of G. A nucleus can be considered as a set of vertices, and we take all edges among these vertices (induced subgraph) to attain the subgraph. The *size* of a nucleus always refers to the number of vertices, unless otherwise specified. For any set S of vertices, the density of the induced subgraph is $|E(S)|/\binom{|S|}{2}$, where $E(S)$ is the set of edges internal to S. *We ignore any nucleus with less than 10 vertices. Such nuclei are not considered in any of our results.*

For brevity, we present detailed results on only 4 graphs (given in Tab. 7.1): `facebook`, `soc-sign-epinions`, `web-NotreDame`, and `wikipedia-200611`. This covers a variety of graphs, and other results are similar.

6.5.1 The forest of nuclei

We were able to construct the forest of $(3, 4)$-nuclei for all graphs in Tab. 7.1, but only give the forests for `facebook` (Fig. 6.3), `soc-sign-epinions` (Fig. 6.6), and `web-NotreDame` (Fig. 6.7). For the `web-NotreDame` figure, we could not present the entire forest, so we show some trees in the forest that had nice branching. The density is color coded, from blue (density 0) to red (density 1). The nuclei sizes, in terms of

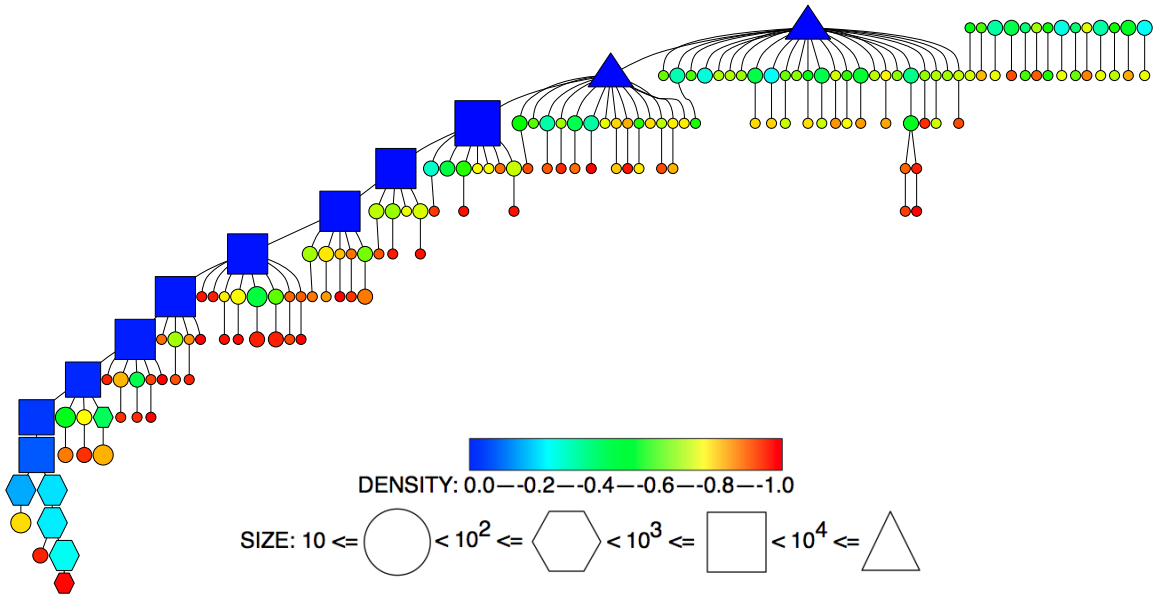


Figure 6.6: $(3,4)$-nuclei forest for `soc-sign-epinions`. There are 465 total nodes and 75 leaves in the forest. There is a clear hierarchical structure of dense subgraphs. Leaves are mostly red (¿ 0.8 density). There are also some light blue hexagons, representing subgraphs of size ≥ 100 vertices with density of at least 0.2.

vertices, are coded by shape: circles correspond to at most 10^2 vertices, hexagons in the range $[10^2, 10^3]$, squares in the range $[10^3, 10^4]$, and triangles are anything larger. The relative size of the shape, is the relative size (in that range) of the set.

Overall, we see that the $(3,4)$-nuclei provide a hierarchical representation of the dense subgraphs. The leaves are mostly red, and their densities are almost always > 0.8. But we obtain numerous nuclei of intermediate sizes and densities. In the `facebook` forest and to some extent in the `web-NotreDame` forest, we see hexagons of light blue to green (nuclei of > 100 vertices of densities of at least 0.2). The branching

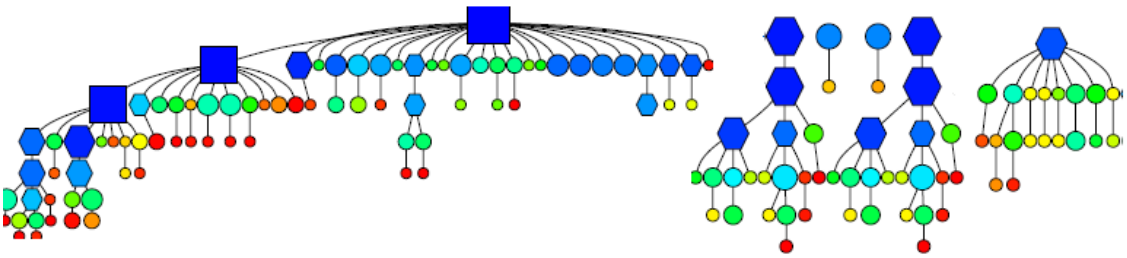
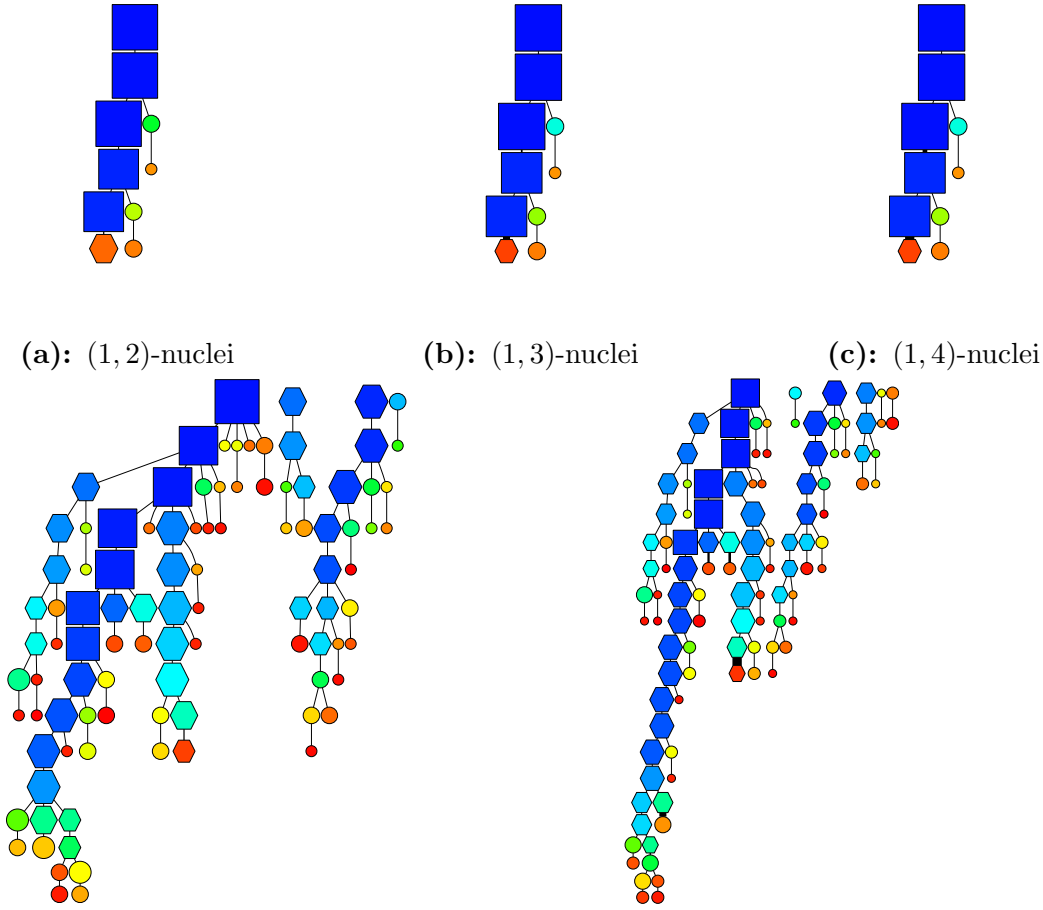


Figure 6.7: Part of the $(3,4)$-nuclei forest for `web-NotreDame`. In the entire forest, there are 2059 nodes and 812 leaves. 79 of the leaves are clique, up to the size of 155. There is a nice branching structure leading to a decent hierarchy.

is quite prominent, and the smaller dense nuclei tend to nest into larger, less dense nuclei. This held in every single $(3,4)$-nucleus forest we computed. This appears to validate the intuition that real-world networks have a hierarchical structure.

The $(3,4)$-nuclei figures provide a useful visualization of the dense subgraph structure. The `web-NotreDame` has a million edges, and it is not possible to see the graph as a whole. But the forest of nuclei breaks it down into meaningful parts, which can be visually inspected. The overall forest is large (about 2000 nuclei), but the nesting structure makes it easy to absorb. We have not presented the results here, but even the `wikipedia-200611` graph of 38 million edges has about a forest of only 4000 nuclei (which we were able to easily visualize by a drawing tool).

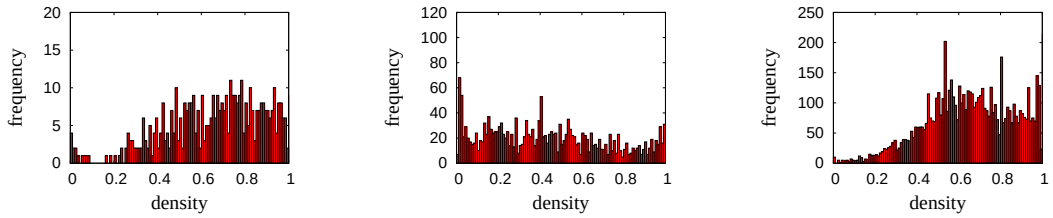
Other choices of r, s for the nuclei do not lead to much branching. We present all nucleus trees for $r < s \leq 4$ for the `facebook` graph in Fig. 6.8 (except $(3,4)$ which is given in Fig. 6.3). Clearly, when $r = 1$, the nucleus decomposition is boring. For $r = 2$, some structure arises, but not as dramatic of Fig. 6.3. Results vary over graphs,



(a): $(1,2)$-nuclei **(b):** $(1,3)$-nuclei **(c):** $(1,4)$-nuclei

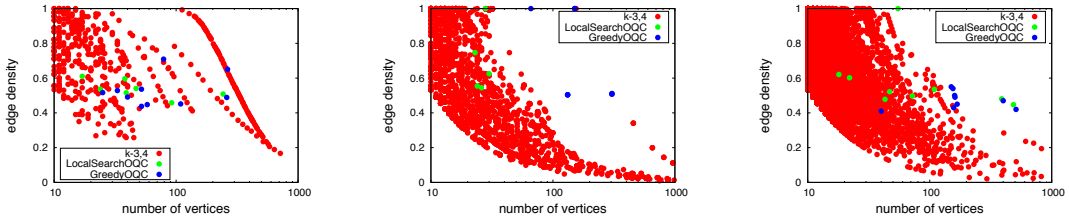
(d): $(2,3)$-nuclei **(e):** $(2,4)$-nuclei

Figure 6.8: (r,s)-nuclei forests for `facebook` when $r < s \leq 4$ (Except $(3,4)$, which is given in Fig. 6.3). For $r = 1$, trees are more like chains. Increasing s results in larger number of internal nodes, which are contracted in the illustrations. There is some hierarchy observed for $r = 2$, but it is not as powerful as $(3,4)$-nuclei, i.e., branching structure is more obvious in $(3,4)$-nuclei.



(a) `soc-sign-epinions` (b) `web-NotreDame` (c) `wikipedia-200611`

Figure 6.9: Density histograms for nuclei of three graphs. x-axis (binned) is the density and y-axis is the number of nuclei (at least 10 vertices) with that density. Number of nuclei with the density above 0.8 is significant: 139 for `soc-sign-epinions`, 355 for `web-NotreDame`, and 1874 for `wikipedia-200611`. Also notice that, the mass of the histogram is shifted to right in `soc-sign-epinions` and `wikipedia-200611` graphs.



(a) `soc-sign-epinions` (b) `web-NotreDame` (c) `wikipedia-200611`

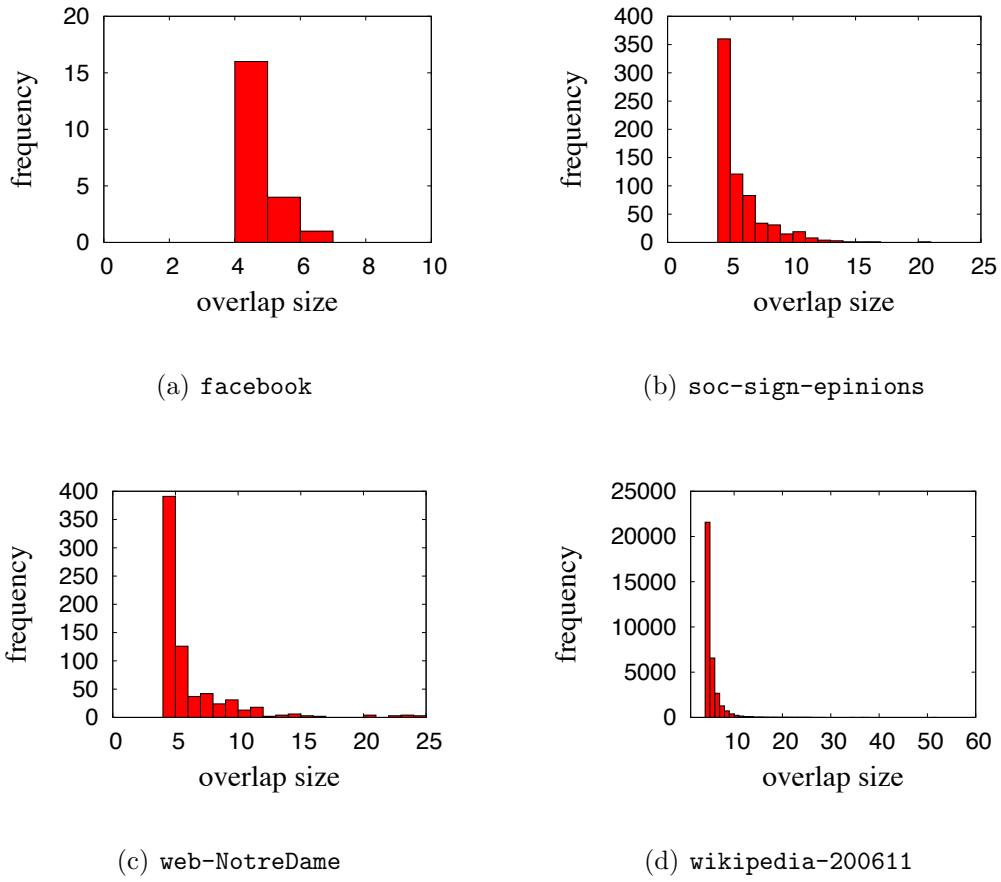
Figure 6.10: Density vs. size plots for nuclei of three graphs. State-of-the-art algorithms are depicted with OQC variants, and they report one subgraph at each run. We ran them 10 times to get a general picture of the quality. Overall, $(3, 4)$-nuclei is very competitive with the state-of-the-art and produces many number of subgraphs with high quality and non-trivial sizes.

but for $r = 1$, there is pretty much just a chain of nuclei. For $r = 2$, some graphs show more branching, but we consistently see that for $(3, 4)$-nuclei, the forest of nuclei is always branched.

6.5.2 Dense subgraph discovery

We plot the density histograms of the $(3, 4)$-nuclei for various graphs in Fig. 6.9. The x-axis is (binned) density and the y-axis is the number of nuclei (all at least 10 vertices) with that density. It can be clearly observed that we find many non-trivial dense subgraphs. It is surprising to see how many near cliques (density > 0.9) we find. We tend to find more subgraphs of high density, and other than the `web-NotreDame` graph, the mass of the histogram is shifted to the right. The number of subgraphs of density at least 0.5 is in the order of hundreds (and more than a thousand for `wikipedia-200611`).

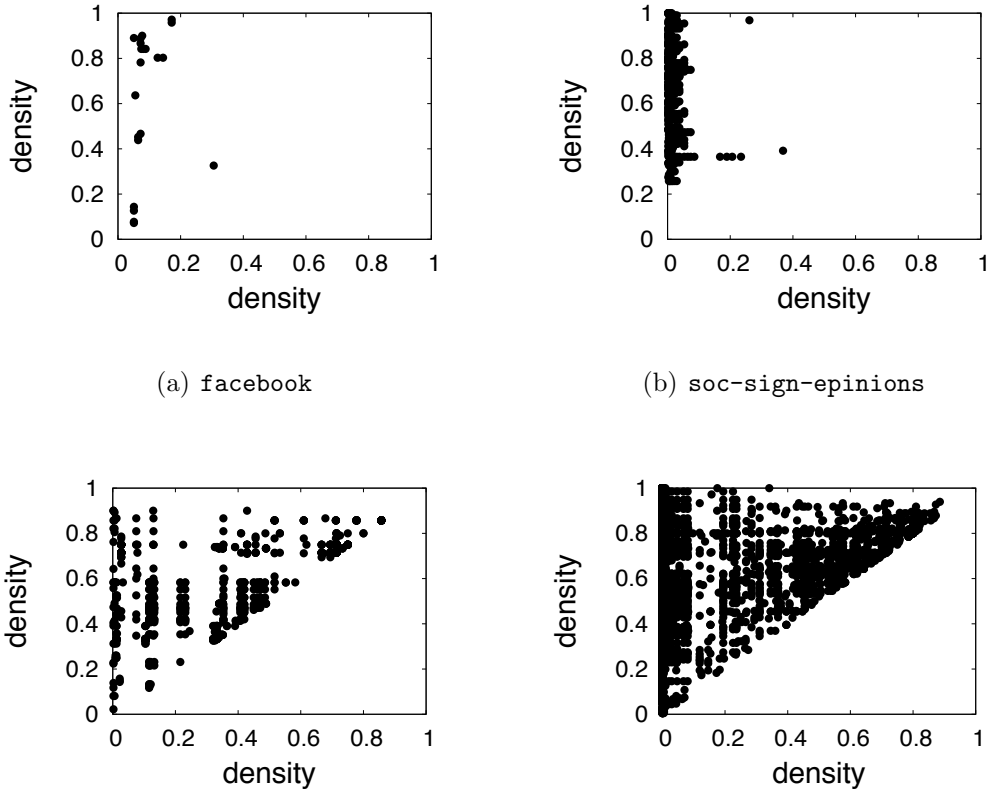
An alternate presentation of the dense subgraphs is a scatter plot of all $(3, 4)$-nuclei with size in vertices versus density. This is given in Fig. 6.2 and Fig. 6.10, where the red dots correspond to the nuclei. We see that dense subgraphs are obtained in all scales of size, which is an extremely important feature. Nuclei capture more than just the densest (or high density) subgraphs, but find large sets of lower density (say around 0.2). Note that 0.2 is a significant density for sets of hundreds of vertices.



(a) `facebook` (b) `soc-sign-epinions`

(c) `web-NotreDame` (d) `wikipedia-200611`

Figure 6.11: Histograms over non-trivial overlaps for $(3, 4)$-nuclei. Child-ancestor intersections are omitted. Overlap size is in terms of the number of vertices. Most overlaps are small in size. We also observe that $(2, s)$-nuclei give almost no overlaps.



(a) `facebook` (b) `soc-sign-epinions`

(c) `web-NotreDame` (d) `wikipedia-200611`

Figure 6.12: Overlap scatter plots for $(3, 4)$-nuclei. Each axis shows the edge density of a participating nucleus in the pair-wise overlap. Larger density is shown on the y-axis. $(3, 4)$-nuclei is able to get overlaps between very dense subgraphs, especially in `web-NotreDame` and `wikipedia-200611`. In `wikipedia-200611` graph, there are 1424 instances of pair-wise overlap between two nuclei, where each nucleus has the density of at least 0.8.

Comparisons with previous art

How does the quality of dense subgraphs found compare to the state-of-the-art? In the scatter plots of Fig. 6.2 and Fig. 6.10, we also show the output of two algorithms of [170] in green and blue. The idea of [170] is to approximate *quasi-cliques*, and their result provides two every elegant algorithms for this process. (We collectively refer to them as OQC.) OQC algorithms only give a single output, so we performed multiple runs to get many dense subgraphs. This is consistent with what was done in [170]. OQC algorithms clearly beat previous heuristics and it is fair to say that [170] is the state-of-the-art.

The $(3, 4)$-nucleus decomposition does take significantly longer than the algorithms of [170]. But we always get much denser subgraphs in all runs. Moreover, the sizes are comparable if not larger than the output of [170]. Surprisingly, in `facebook` and `soc-sign-epinions`, some of the best outputs of OQC are very close to $(3, 4)$-nuclei. Arguably, the $(3, 4)$-nuclei perform worst on `wikipedia-200611`, where OQC find some larger and denser instances than $(3, 4)$-nuclei. Nonetheless, the smaller $(3, 4)$-nuclei are significantly denser. We almost always can find fairly large cliques.

In Tab. 7.1, we consider the OQC output vs $(3, 4)$-nuclei for all graphs. Barring 4 instances, there is a $(3, 4)$-nucleus that is larger and denser than the OQC output. In all cases but one (adjnoun), there is a $(3, 4)$-nucleus of density (of non-trivial size) higher than the the OQC output. The nuclei have the advantage of being the output of a fixed, deterministic procedure, and not a heuristic that may give different outputs

on different runs. We mention that OQC algorithms have a significant running time advantage over finding $(3,4)$-nuclei, for a single subgraph finding.

6.5.3 Overlapping nuclei

A critical aspect of nuclei is that they can overlap. Grappling with overlap is a major challenge when dealing with graph decompositions. We believe one of the benefits of nuclei is that they naturally allow for (restricted) overlap. As mentioned earlier, no two (r,s)-nuclei can contain the same K_r. This is a significant benefit of setting $r=3, s=4$ over other choices.

In Fig. 6.11, we plot the histogram over non-trivial overlaps for $(3,4)$-nuclei. (We naturally do not consider a child nucleus intersecting with an ancestor.) For a given overlap size in vertices, the frequency is the number of pairs of $(3,4)$-nuclei with that overlap. This is shown for four different graphs. The total number of pair-wise overlaps (the sum of frequencies) is typically around half the total number of $(3,4)$-nuclei. We observed that the Jaccard similarities are less than 0.1 (usually smaller). This suggest that we have large nuclei with some overlap.

There are bioinformatics applications for finding vertices that are present in numerous dense subgraphs [87]. The $(3,4)$-nuclei provide many such vertices. In Fig. 6.12, we give a scatter plot of all intersecting nuclei, where nuclei are indexed by density. For two intersecting nuclei of density $\alpha > \beta$, we put a point (α, β). We only plot pairs where the overlap is at least 5 vertices. Especially for `web-NotreDame` and `wikipedia-200611`, we get significant overlaps between dense clusters.

In contrast, for all other settings of r, s, we get almost no overlap. When $r = 2$, nuclei can only overlap at vertices, and this is too stringent to allow for interesting overlap.

6.5.4 Runtime results

Tab. 7.1 presents the runtimes in seconds for the entire construction. To provide some context, we describe runtimes for varying choices of r, s. For $r = 1, s = 2$ (k-cores), the decomposition is linear and extremely fast. For the largest graph (`wikipedia-200611`) we have, with $39M$ edges, it takes only 4.26 seconds. For $r = 2, s = 3$ (trusses), the time can be two orders of magnitude higher. And for $(3, 4)$-nuclei, it is an additional order of magnitude higher. Nonetheless, our most expensive run took less than an hour on the `wikipedia-200611` graph, and the final decomposition is quite insightful. It provides about 6000 nuclei with more than 10 vertices, most of them of have density of at least 0.4. The algorithms of [170] take roughly a minute for `wikipedia-200611` to produce *only one* dense subgraph.

The theoretical running time analysis of Thm. 11 gives a running time bound of $\sum_v c_3(v)d(v)$. In Tab. 7.1, we show this value for the various graphs. In general, we note that this value roughly correlates with the running time. For graphs where the running time is in many minutes, this quantity is always in the billions. For the large

wiki graphs where the $(3, 4)$-nucleus decomposition is most expensive, this is in the trillions.

6.5.5 Application on protein-protein interaction networks

Understanding the function, expression and regulation of the proteins encoded by an organism is crucial in the field of genetics. Previously it has been observed that proteins do not act in an isolated way, but rather interact with each other to be involved in the same cellular processes. Finding the dense regions in the protein-protein interaction (PPI) networks can yield meaningful set of proteins involving in a specific cellular process. We apply our algorithms on PPI networks to find those dense regions and investigate their quality using the ground-truth information in PPI networks. For this purpose, we use Gene Ontology (GO) Enrichment analysis [14] to evaluate the reported set of proteins. We made use of AmiGO tool [37], which takes a set of proteins as an input and reports several results indicating the quality of that set of proteins. Background frequency is the number of genes annotated to a GO term in the entire background set, while sample frequency is the number of genes annotated to that GO term in the input set of proteins. P-value is the probability or chance of seeing at least x number of genes out of the total n genes in the list annotated to a particular GO term, given the proportion of genes in the whole genome that are annotated to that GO Term. That is, the GO terms shared by the genes in the user's list are compared to the background distribution of annotation. The closer the

p-value is to zero, the more significant the particular GO term associated with the group of genes is.

We selected the Mus_musculus PPI network, having 8573 proteins and 21003 interactions, obtained from BioGRID database [168] and chose biological process ontology. We ran our $(3, 4)$-nucleus decomposition and the *Local Search OQC* algorithm, given in [170], (multiple times) and selected the subgraphs having highest edge density with the size of at least 10 vertices. $(3, 4)$-nucleus decomposition reports a subgraph with 13 vertices and 70 edges (0.89 edge density) and *Local Search OQC* results in a subgraph with 10 vertices and 27 edges (0.60 edge density). Note that $(3, 4)$-nucleus decomposition is able to report a larger subgraph with higher edge density. The UniProt ids of the proteins in the subgraph reported by *Local Search OQC* algorithm are Cyld, Ikbkb, Ikbkg, Rhoa, Ripk1, Rnf31, Sqstm1, Traf6, UBC, and Ubc. For this set of proteins, AmiGO reports that there are 62 GO terms for which the p-value is non-zero. Most significant p-value ($2.056e-09$) is observed for the regulation of I-kappaB kinase/NF-kappaB signaling process [123]. Sample frequency for this process is 7, whereas the background frequency is 182. On the other side, $(3, 4)$-nucleus decomposition gives a higher quality set of proteins. UniProt ids of them are Esrrb, Hdac1, Hdac2, L3mbtl2, Mbd3, Mta1, Mta2, Nanog, Pou5f1, Rbbp7, Rnf2, Sall4, and Tfcp2l1. There are 152 GO terms for those proteins where the p-value is non-zero, and the most significant p-value is $4.937e-10$, observed for the negative regulation of nucleic acid-templated transcription, which is any process that stops, prevents or

reduces the frequency, rate or extent of nucleic acid-templated transcription. Sample frequency for this process is 11, and the background frequency is 960. These results suggest that our $(3, 4)$-nucleus decomposition is able to find more meaningful set of proteins than the *Local Search OQC* algorithm, and can be effectively used for PPI network mining applications.

6.6 Further directions

The most important direction is in the applications of nucleus decompositions. We are currently investigating bioinformatics applications, specifically protein-protein and protein-gene interaction networks. Biologists often want a global view of the dense substructures, and we believe the $(3, 4)$-nuclei could be extremely useful here. In our preliminary analyses, we wish to see if the nuclei pick out specific functional units. If so, that would provide strong validation of dense subgraph analyses for bioinformatics.

It is natural to try even larger values of r, s. Preliminary experimentation suggested that this gave little benefit in either the forest or the density of nuclei. Also, the cost of clique enumeration becomes forbiddingly large. It would be nice to argue that $r = 3, s = 4$ is a sort of sweet spot for nucleus decompositions. Previous theoretical work suggests that any graph with a sufficient triangle count undergoes special "community-like" decompositions [78]. That might provide evidence to why triangle based nuclei are enough.

A faster algorithm for the $(3, 4)$-nuclei is desirable. Clique enumeration is a well-studied problem [33], and we hope techniques from these results may provide ideas here. Of course, as we said earlier, any method based on storing 4-cliques is infeasible (space-wise). We hope to devise a clever algorithm or data structure that quickly determines the 4-cliques that a triangle participates in.

Last but not least, we seek for incremental algorithms to maintain the (r, s)-nuclei for a stream of edges. There are existing techniques for streaming k-core algorithms [145] and we believe that similar methods can be adapted for (r, s)-nuclei maintenance.

Chapter 7: Streaming Overlapping Community Detection

In many application domains, graphs are used to represent relationships between people, systems, and the physical world. Data analytics performed on these graphs can bring new business insights and improve decision-making. For instance, the graph structure may represent the relationships in a social network, where finding communities in the graph [102] can facilitate targeted advertising. As another example, in the Telecommunications domain, call details records can be used to capture the call relationships between people [129], and locating closely connected groups of people can help generate promotions.

As these examples illustrate, a fundamental graph analytic is *community detection*. We can define a community within a graph as a set of vertices that exhibit high *cohesiveness* and low *conductance*. High cohesiveness means that the vertices in the community have relatively high number of edges connecting them, and low conductance means that the vertices in the community have relatively small number of edges going outside of the community.

7.1 Introduction

Communities in social networks have two key characteristics. The first is that communities are *overlapping*, as different communities can have common users. This is a typical scenario, as a single user can be involved in different communities, such as co-workers, friends, and family. The second is that communities are *dynamic*. They evolve as a result of the continuous interactions between people. These interactions can result in the addition/removal of new/existing relationships in the network. For instance, the follower-followee graph of Twitter [173] is highly active, with millions of updates to the graph structure every day. This number is even higher if we consider the mention graph of Twitter. It is also common to analyze the graph over a recent time window, such as the mention graph of Twitter over the last week. In such scenarios, both insertions and removals are equally frequent.

In this chapter, we present SONIC—an algorithm to detect overlapping communities on dynamic graphs in a *streaming* manner. Upon each edge insertion or removal, we *incrementally* maintain the overlapping communities. This way, the communities are updated more efficiently and without the need for periodic re-computations that are typically performed in batch. SONIC maintains multiple community ids for each vertex and updates these ids upon edge insertions and removals. By doing so, it can answer any query for the communities of a given vertex (or a set of vertices) by a simple traversal of the community ids.

SONIC adopts the find-and-merge style of community detection. In find-and-merge style algorithms [47, 139], local communities of each vertex are found first, as part of the *find step*. These local communities are then merged into global communities based on a configurable merge condition, as part of the *merge step*. SONIC uses the label propagation algorithm to detect local communities during the find step. In the label propagation algorithm [138], each vertex is initially assigned a unique id. Then each vertex gets the most commonly used id of its neighbors. This procedure continues either for a specified number of rounds or until there are no changes in ids. After that, SONIC merges local communities based on a given merge factor. Differently than earlier works, SONIC relies on an *incremental merge* step to avoid rebuilding global communities from scratch and instead performs a partial re-merge.

SONIC faces several challenges in tackling the streaming overlapping community detection problem. First, in a streaming setup, the number of updates are very high, yet many of these updates are not sufficiently important by themselves to result in any change in the global community structure. Fully processing each one of these updates will unnecessarily increase the cost of the solution. SONIC addresses this problem by detecting updates that are *significant* via a fast procedure that involves re-adjusting only the local communities. SONIC initiates the merge process only for the significant updates, effectively reducing the cost of an edge update. Second, the merge step is non-trivial to perform incrementally, which led earlier work on find-and-merge style of algorithms to use a non-incremental merge [47]. SONIC solves this problem by

maintaining the global communities as a collection of sub-communities. Upon an edge insertion/removal, it detects the global communities whose sub-communities are impacted, dissolves such global communities, and regenerates the global community structure by a partial merge. Finally, even an incremental merge algorithm can be costly to execute when the local changes cascade to bring about a major change in the global communities. To address this issue, SONIC incorporates two alternative merge strategies: (*i*) a *min-hash* based merge, and (*ii*) an *inverted-index* based merge.

In summary, this chapter makes the following contributions:

- The SONIC algorithm for incremental overlapping community detection over dynamic graphs with streaming updates.
- A technique to detect significant changes in local community structures to avoid a costly merge, unless a local community change is likely to cause a global community change.
- Inverted-index and min-hash based techniques to further accelerate the incremental merge used in SONIC.
- An experimental evaluation of SONIC on real-world and synthetic data sets, with respect to quality and running time performance.

The rest of this chapter is organized as follows. Section 7.2 gives an overview of related work. Section 7.3 gives the background on basic techniques from the literature we rely on. Section 7.4 lists fundamental theoretical properties that we leverage for

developing the SONIC algorithm. Section 7.5 describes the base version of the SONIC algorithm with an illustrative example. Section 7.6 presents several improvements over base SONIC, such as the significant change detection, the min-hash based merge, and the inverted-index based merge algorithms. Section 7.7 presents our experimental evaluation and Section 7.8 concludes the chapter.

7.2 Related Work

Vast amount of work has been done on community detection and various aspects of the problem have been studied in the literature. Fortunato [63] covers all the popular techniques to find communities in complex networks. Leskovec et al. [106] compares different community detection algorithms empirically. Significant number of spectral methods are based on the *modularity* metric proposed by Newman [130]. There are also information theoretic approaches to discover community structure of networks. Particularly, Infomap [141] is currently one of the best performing non-overlapping community detection algorithms. As an alternative technique to community detection, past works have proposed community search, where the communities are detected locally based on given query vertices. This idea first appeared in [83] and was further investigated in [167] and [133]. Recently, Cui et al. [49] proposed online search of overlapping communities based on clique adjacency graph of networks.

In our work, we are particularly interested in (i) overlapping community detection techniques, and (ii) dynamic methods that can handle streaming updates.

Overlapping. Community detection in social networks is different from the classical clustering and partitioning problems in which the identified clusters/partitions do not overlap with each other (i.e., each vertex belongs to a single cluster/partition). In contrast, communities overlap with each other in social networks. Palla et al. [134] showed that most real networks have overlapping community structure. They also introduced a percolation based method to detect overlapping communities. A recent survey [184] summarizes most of the existing overlapping community detection algorithms and categorizes them into five classes: (1) Clique percolation [134], (2) Link partitioning [5], (3) Local expansion and optimization [103, 181], (4) Fuzzy detection [77, 176, 188] and (5) Agent-based algorithms [138]. Among them, Hierarchical Link Clustering (HLC) [5] is a popular approach due to its simplicity. It partitions the links instead of vertices to explore the overlapping community structure. In the local expansion and optimization based algorithms category, Whang et al. [181] recently proposed an algorithm which finds good seeds and then expands them using a personalized PageRank clustering procedure. Fuzzy detection algorithms measure the strength of association between vertices and communities. They use membership vectors to determine the strength of these associations and determine communities according to these vectors. SONIC belongs to a new class of overlapping community detection algorithms, which we call *find-and-merge*.

Communities in large graphs can have different granularities. For example, given the co-author graph, one possible community would be "people working on databases".

However such a community is very coarse-grained and not so effective in many scenarios. Instead, finding the communities based on vertices provide finer granularity and more focused information. For example, a query like "people working with Prof. X on graph processing" could be more interesting and useful. Recent studies [74] also show that vertex based approaches provide better communities in terms of widely accepted metrics, such as conductance. Thus, finding fine-grained communities around vertices, which provide grounds for answering queries like "Which communities contain the vertex u?" and "Which communities include the vertices $u_1, \cdots, u_n$?", is highly valuable. Motivated by this observation, Rees and Gallagher [139] and Coscia et al. [47] developed a new class of overlapping community detection algorithms that follow a bottom-up approach, which we refer to as find-and-merge type of algorithms. For instance, DEMON [47] first finds the local communities of each vertex. It then merges these local communities into global communities based on a configurable merge condition, as part of the merge step. The SONIC algorithm we describe in this chapter adopts a similar approach. Different from [47] and [139], SONIC is an incremental find-and-merge algorithm that can handle streaming updates.

Evolutionary Clustering. Several researchers have investigated evolutionary and dynamic community detection algorithms. Evolutionary clustering techniques capture how the clusters change as a function of time [38, 96, 110]. Previous work have focused on different aspects of evolutionary clustering, such as the formalization of

smoothness between clustering results in recent snapshots [43], and detecting the behavioral patterns of temporal interaction networks [15]. Detailed survey and analysis on evolutionary graph clustering techniques can be found in [28], and in [4]. However, these techniques focus on temporal evolution at a coarser level and do not address the issues of incremental processing and streaming updates to maintain the community structure of the graph.

Streaming. As an example of incremental community detection algorithm, Xie et al. [183] proposed an incremental version of label propagation which can be used to find non-overlapping communities in evolving networks. SONIC also uses an incremental label propagation algorithm, but only as an initial step to find the local non-overlapping communities to be merged later into global overlapping communities. There are also several prior works focusing on streaming dense subgraph detection, which is a problem similar to community detection. Agarwal et al. [2] and Angel et al. [12] present algorithms for real-time discovery of events and stories from micro blog streams. They model the events and stories as dense subgraphs and track their evolution in a streaming fashion. This chapter focuses on an algorithm for *incremental, overlapping community detection.*

7.3 Background

In this section, we provide the necessary background on concepts and algorithms that are relevant to our work.

Ego-minus-ego network. Let G be an undirected and unweighted graph. For a vertex u, $N(u)$ is the set of neighbors of vertex u in graph G. The *ego-network* [66] of u is the vertex induced sub-graph of G that consists of the vertices $\{u\} \cup N(u)$. Subtracting u and the edges incident upon it from the ego-network results in the *ego-minus-ego network* [47], which we formally define as follows:

Definition 13. *Ego-minus-ego network of vertex u, or EmEn(u) in short, in graph $G{=}\langle V, E\rangle$ is the subgraph $G'{=}\langle V', E'\rangle$, where $V'{=}N(u)$ and $E' \subseteq E$ is the set of edges (v, w) such that $v, w \in V'$. Vertex u is the* center *of EmEn(u).*

Find-and-merge style overlapping community detection. This style of community detection algorithms can compute overlapping global communities from local communities [47, 139]. The basic idea is to find local communities in each *EmEn* via a non-overlapping community detection algorithm, such as label propagation, and then add the center of the *EmEn* to each one of the local communities found. After the find step is complete, the algorithm merges the local communities to construct the global ones. The merge is performed based on a *merge factor* parameter, denoted by β, where $\beta \in [0, 1]$. During the merge step, two communities are merged if at least β fraction of the smaller community resides in their intersection. If A and B are two communities, the merge condition is given by $|A \cap B|/\min(|A|, |B|) \geq \beta$. This is the well-known *overlap similarity* metric. The merge process continues until no additional merges can be computed.

Incremental local community detection via label propagation. Our non-overlapping local community detection algorithm of choice for the find step is label

propagation [138]. This algorithm works in multiple rounds. Initially, each vertex is assigned a unique id. Then at each round, each vertex is assigned the id of the most commonly used id of its neighbors. Ties are broken randomly. This procedure is performed continuously either for a specified number of rounds or until there is no change in the ids. When an edge (u, v) is inserted/removed into/from the graph, we perform incremental label propagation starting with vertices u and v. That is, we assign their ids to the most commonly used id of their neighbors. If this results in a change in the id values of u or v, then we apply the same procedure to all neighbors of the vertex whose id has changed, and continue the procedure recursively.

Observation 7. *After the merge step, each local community takes part in one and only one global community. Therefore, the global communities are a partitioning of the local ones.*

Definition 14. *Local communities of a vertex $u \in V$, denoted by $LC(u)$, is the set of communities found by a non-overlapping community detection algorithm in $EmEn(u)$, augmented by the vertex u itself. Global communities, denoted by GC, are the set of communities constructed by merging the local communities of all vertices in the graph. Finally, the set of local communities that were merged to form a global community $g \in GC$ is denoted as $MC(g)$.*

7.4 Observations

In this section we list fundamental observations that we rely on for developing the SONIC algorithm.

Theorem 12. *Given a graph $G=\langle V, E\rangle$, if we insert or remove an edge (u, v), only the EmEns of u, v, and mutual neighbors of u and v change.*

Proof. Consider the insertion case first. The *EmEn*s of u and v change, as $EmEn(u)$ will now contain v, and $EmEn(v)$ will contain u. *EmEn*s of common neighbors, that is $N(u) \cap N(v)$, will change as well, since for a vertex $w \in N(u) \cap N(v)$, the edge

(u, v) will now be contained in $EmEn(w)$. For any other vertex $x \notin N(u) \cap N(v)$ that is not equal to u or v, it is easy to see that $EmEn(x)$ cannot change. Assume that it does. This change cannot involve a new vertex, say u', being added into $EmEn(x)$, as that would require (x, u') to be a new edge, and since $x \neq u$ and $x \neq v$, this is a contradiction. The other possibility is that a new edge is added into $EmEn(x)$, which must be (u, v). But then we have $u \in N(x)$ and $v \in N(x)$, which means $x \in N(u) \cap N(v)$, contradicting the initial assumption.

The removal proof follows trivially. Assume that the *EmEn* of some vertex other than u or v or a common neighbor of them has changed. Then inserting the removed edge back will also result in a change for the *EmEn* of that vertex, which contradicts the first part. By the same logic, *EmEn*s of u, v, and vertices in $N(u) \cap N(v)$ will change as a result of the removal. □

Corollary 6. *If u and v have no mutual neighbors and if an edge is inserted between them, EmEns of u and v will gain an* unconnected *vertex v and u, respectively.*

Theorem 13. *Given a set of communities, the result of merging them based on an overlap similarity threshold of $\beta < 1$ is sensitive to the merge order.*

Proof. Assume that the merge-order is not important and same solution is obtained for any given set and β coefficient. Say that we have three communities $A = \{1, 2, 3\}, B = \{3, 4\}$ and $C = \{4, 5, 6\}$ and $\beta = 0.5$. If we merge A and B first, then resulting communities will be $AB = \{1, 2, 3, 4\}$ and $C = \{4, 5, 6\}$. However, if B and C are merged first, then the resulting communities are $A = \{1, 2, 3\}$ and $BC = \{3, 4, 5, 6\}$. Therefore, the merge result is sensitive to the merge order. □

Definition 15. *For a find-and-merge type of algorithm, any set of communities that is reached via some merge order, such that no further merges are possible between any two communities, is a* valid *solution.*

Theorem 14. *Given the set of global communities GC, and the local communities $MC(g), \forall g \in GC$; if a local community $l \in MC(g')$ that is part of a global community $g' \in GC$ changes, then merging the set of local communities within $MC(g')$ plus the remaining global communities $GC \setminus \{g'\}$ will give a valid solution.*

Proof. Proof follows from Definition 15. Since any merge order is valid as long as no further merges are possible, we change the merge order of communities to get a valid solution. When we merge the communities in $\bigcup_{g \in GC \setminus \{g'\}} MC(g)$, we will get $GC \setminus \{g'\}$. Then, merging $GC \setminus \{g'\}$ with $MC(g')$ will give us GC. In other words, a from-scratch merge can have such an order that results in first generating the global communities in $GC \setminus \{g'\}$ and then merging in the local communities in $MC(g')$. This is exactly what the incremental merge performs. □

Corollary 7. *Given the set of global communities GC, and the local communities* $MC(g), \forall g \in GC$*; if a set of local communities L that are contained within a set of global communities* $\{g_j\} \subset GC$ *change, then merging the local communities within* $\bigcup_j MC(g_j)$ *plus the remaining global communities* $GC \setminus \{g_j\}$ *will give a valid solution.*

7.5 The SONIC Algorithm

In this section, we introduce the SONIC algorithm for incremental overlapping community detection.

7.5.1 An Overview

SONIC is a find-and-merge style of community detection algorithm with a particular focus on incremental processing, as it aims to support streaming updates. SONIC's algorithmic steps are as follows:

1. Find the vertices whose local communities are impacted upon an edge insertion or removal.
2. Perform incremental, non-overlapping local community detection to update the local communities of the impacted vertices.
3. Detect significant changes and terminate if a change in impacted local communities is found to be insignificant.
4. Incrementally merge communities and update the global communities.
   (a) Determine a small set of communities to be merged.
   (b) Perform recursive merge of these in an efficient manner.

For local community detection SONIC uses the incremental label propagation algorithm. The significant change detection capabilities of SONIC are described in Section 7.6.1. In the rest of this section, we focus on the core capability of SONIC: determining the set of communities to be merged, which is significantly smaller in size compared to the entire set of local communities. The efficient procedures used by SONIC to merge these communities are described in Sections 7.6.2 and 7.6.3.

7.5.2 SONIC Core

SONIC handles edge insertions/removals by locating the impacted local communities, dissolving the global communities that contain them, replacing the impacted local communities with their updated versions, and performing a partial re-merge to create the new set of global communities.

In particular, when a new edge is inserted/removed, some local communities, say L, are changed. Assume that these changed local communities are part of some set of global communities, denoted by $C(L) = \{g : \exists l \in L \text{ s.t. } l \in MC(g) \wedge g \in GC\}$. Further assume that the local communities are replaced with their updated versions, say L'. By Corollary 7, SONIC regenerates the new global communities by merging L' and $GC \setminus C(L)$, that is:

$$GC \leftarrow \text{merge}(L', GC \setminus C(L))$$

To facilitate this merge, SONIC keeps track of which local communities were merged to construct each global community. For this purpose, it keeps the following additional data structures:

1. GC: For each vertex, the global community ids associated with that vertex.
2. LC: For each vertex, the local community ids associated with that vertex.
3. For each global community, the local community ids that constitute it.
4. For each local community, the global community id that it belongs to.

SONIC maintains global communities at each vertex to speedup the merge process. Each vertex u can be part of at most $|N(u)|$ local/global communities, requiring $\mathcal{O}(|E|)$ storage for global community ids. In practice, the number of global communities a vertex u belongs to is considerably smaller than the upper bound of $|N(u)|$. The total number of local communities, $|LC|$, can be at most $2 \cdot |E|$. Thus, keeping the global communities with their constituent local communities also takes $\mathcal{O}(E)$ space. Again, this happens only for the highly unlikely scenario of each edge belonging to a different local community. Our evaluation shows that the space overhead of the data structures kept by SONIC corresponds to a small constant times the number of edges (see Section 7.7).

Given a graph G and global communities GC, if an edge (u, v) is inserted/removed, SONIC updates the global communities using Algorithm 25. First, it performs the insertion/removal. Then it checks the mutual neighbors of u and v. If there are no

Algorithm 25: SONIC $(G, (u, v), op, \beta, LC, GC)$

Input: G: graph, (u, v): updated edge, op: operation ('i'/'r') LC: local communities, GC: global communities, β: merge factor

1 **if** $op =$ 'i' **then** $G \leftarrow G \cup \{(u, v)\}$ ▷ Insertion operation
2 **else** $G \leftarrow G \setminus \{(u, v)\}$ ▷ Removal operation
3 **if** $N(u) \cap N(v) = \emptyset$ **then** ▷ No common neighbors
4 **return** ▷ No update needed
5 $S \leftarrow \{u, v\} \cup (N(u) \cap N(v))$ ▷ Vertices with changed *EmEn*
6 $R \leftarrow \{u \mapsto LC(u) : u \in S\}$ ▷ Local comm. of vertices in S
7 IncrLabelPropagation(S, LC) ▷ Update local comm.
▷ Find the removed local communities
8 $L^- \leftarrow \{l : l \notin LC(u) \wedge l \in R[u] \wedge u \in S\}$
▷ Find the added local communities
9 $L^+ \leftarrow \{l : l \in LC(u) \wedge l \notin R[u] \wedge u \in S\}$
▷ Find the set of global communities to be dissolved
10 $C \leftarrow \{g : \exists l \in L^-$ s.t. $l \in MC(g) \wedge g \in GC\}$
▷ Dissolve comm. in C, remove non-existing local comms.
11 $\mathcal{F} \leftarrow \{MC(g) \setminus L^- : g \in C\}$
12 $GC \leftarrow GC \setminus C$ ▷ Remove dissolved global comms.
13 Merge$(G, \beta, GC, \mathcal{F}, L^+)$ ▷ Re-merge into global comms.

mutual neighbors, we know that only u's and v's *EmEn*s change, and v and u are added as unconnected vertices to the *EmEn*s of u and v, respectively (see Corollary 6). In this case, it assumes that there is no change in the local community structure of u and v, and therefore it performs no further operation to update GC (line 4).

When u and v have mutual neighbors, the *EmEn*s of u, v, and their mutual neighbors change (see Theorem 12). Thus, SONIC collects these vertices in a set S (line 5). Next, it creates a map R to keep the set of local communities associated with the vertices whose *EmEn*s has changed (line 6). This map temporarily keeps the old local communities so that they can be compared to the new ones that are formed

after the insertion/removal. The next step computes the new local communities by running the incremental label propagation algorithm.

After updating the local communities, SONIC finds the set of old local communities that no longer exist, denoted by L^- (line 8); as well as the set of newly created local communities, denoted by L^+ (line 9). Using L^-, it finds the set of global communities to be dissolved, denoted by C (line 10). These are the global communities that currently contain nonexistent local communities. SONIC then dissolves each such global community g by converting it into a set of local communities $MC(g)$ and removing the nonexistent local communities, that is $MC(g) \setminus L^-$. The end result is a *set of local community sets* (denoted by $\mathcal{F}$), where each one of the local community sets represents a dissolved global community (line 11).

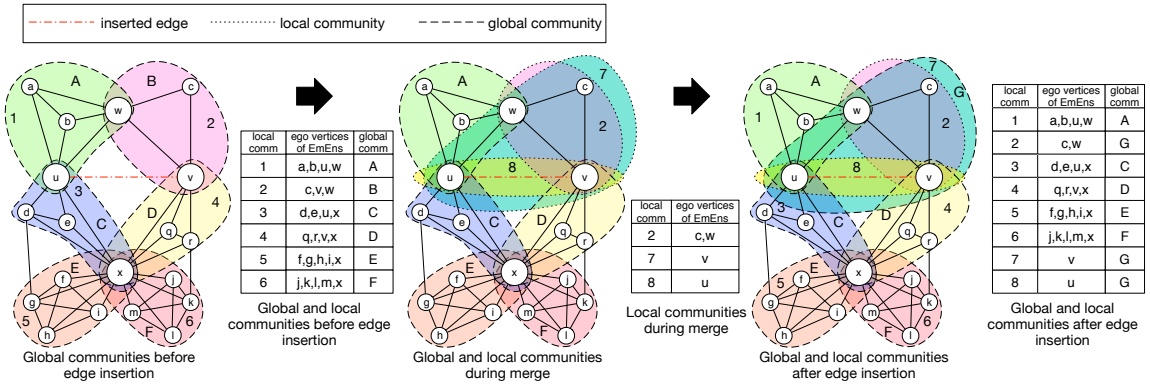
Finally, SONIC merges the set of dissolved global communities $\mathcal{F}$ and the new set of local communities L^+ together with the global communities that are kept intact, that is $GC \setminus C$, using the merge factor β. In the base version of SONIC, we apply the merge operation given in Algorithm 26, named INCNAIVE.

The INCNAIVE algorithm consists of three phases. The first one is called the *Distribute New Phase* (lines 1 to 3). In this phase, the algorithm distributes the newly created local communities, $l \in L^+$, over the dissolved communities, $\mathcal{F}$. To do that, it selects a set $F \in \mathcal{F}$ for each l such that there is a community $f \in F$ that overlaps with l, i.e., $f \cap l \neq \emptyset$. The goal here is to assign each new local community to one of the dissolved communities where it is likely to merge with an existing local

community. This heuristic results in a higher number of cheaper merges in the second phase, yet a lower number of more expensive merges in the third phase. Thus, it is effective in reducing the overall merge cost.

The second phase is called the *Regroup Dissolved Phase* (lines 4 to 11). In this phase, the algorithm performs a merge within each dissolved community $F \in \mathcal{F}$, separately. The motivation is that it is highly likely for the local communities that made up a dissolved global community to re-merge. By doing smaller-scale merges that are localized to the dissolved communities, we aim at reducing the overall cost of the merge. During the merge, the algorithm checks each pair of communities, f_i and f_j, where $j > i$, to see if their overlap similarity is above the merge threshold β. If so, it merges f_j into f_i and removes f_j from its belonging set F. After it completes the merge for a dissolved community, it adds the resulting merged communities to a *global merge list*, denoted as L (line 11).

In the last phase, called the *Global Merge Phase* (lines 12 to 20), the algorithm performs a merge between communities in the global merge list, that is L, and the intact global communities, that is GC. Here we check whether any community within the global merge list can be merged with each other (c_i and c_j, $i < j$), as well as with any of the intact global communities (GC). This is why the outer loop's body contains loops that iterate over both L and GC. However, the outer loop only iterates over L, because once all the communities in L are merged with the rest, the only possible comparisons that remain are between the intact global communities, and we are



local comm	ego vertices of EmEns	global comm
1	a,b,u,w	A
2	c,v,w	B
3	d,e,u,x	C
4	q,r,v,x	D
5	f,g,h,i,x	E
6	j,k,l,m,x	F

local comm	ego vertices of EmEns
2	c,w
7	v
8	u

local comm	ego vertices of EmEns	global comm
1	a,b,u,w	A
2	c,w	G
3	d,e,u,x	C
4	q,r,v,x	D
5	f,g,h,i,x	E
6	j,k,l,m,x	F
7	v	G
8	u	G

Figure 7.1: Illustration of the community changes upon an edge insertion. After inserting an edge between u and v, global community B evolves into a bigger global community G.

guaranteed that those cannot merge (as they are intact, and thus known to have overlap similarity of less than β).

7.5.3 Illustrative Example

Figure 7.1 illustrates the community changes when we insert an edge into a sample graph. The example assumes that the merge factor (β) is 1.0, i.e., two communities can merge only if one of them is a subset of the other. The leftmost figure and table show the global and local communities before the edge insertion is performed. Global communities are shown with bold dashed lines and capital letter ids. For example, community A is a global community consisting of vertices a, b, u, and w. Local communities within the *EmEn*s of ego vertices are given in the tables. For example, in the leftmost table, community 2 is a local community belonging to the *EmEn* of c, as well as v and w. The local communities that form a global community can also

be read from the table. This can be achieved by finding all rows that contain the target global community. The local communities corresponding to these rows form the global community. For example, in the rightmost table, the global community G is composed of the local communities 2, 7, and 8. Note that, finding the local communities belonging to a global community does not require a full scan of the table in the implementation. We make use of our data structures to access local communities of a global community in constant time.

When we insert the edge (u, v), we first check the vertices whose *EmEn*s are changed. Based on Algorithm 25, those vertices are u, v and their mutual neighbors w and x (all shown using larger circles in the figure). For each of these vertices, we check if there is a change in their local community structure by performing incremental label propagation (line 7 of Algorithm 25). For vertex w, it turns out that there is no change in the local community structure, because the inserted edge is not strong enough to bind local communities 1 and 2. Similarly, there is no change in the local community structure of vertex x, as the inserted edge cannot cause a connection between the local communities 3 and 4. On the other hand, the local community structures of u and v change. After the removed and newly created local communities are detected (lines 8 and 9 in Algorithm 25), local community 2 of vertex v is the only removed local community in the new local community structure, i.e., $L^- = \{(v, 2)\}$; whereas local communities 7 and 8 are the newly formed communities of vertices v and u, respectively, i.e. $L^+ = \{(v, 7), (u, 8)\}$. Next, we find the global communities

to be dissolved and it turns out that global community B is the only one that must be dissolved, i.e., $C = \{B\}$ (local community 2 belongs to global community B) . Then, we find $\mathcal{F}$ by dissolving B and subtracting local community 2 of v, i.e., $\mathcal{F} = \{\{(c, 2), (w, 2)\}\}$.

Finally, we set $GC = \{A, C, D, E, F\}$ and merge GC, $\mathcal{F}$, and L^+. After performing the first phase of Algorithm 26, we distribute both local communities in L^+ to the only $F \in \mathcal{F}$ and obtain $F = \{(c, 2), (w, 2), (v, 7), (u, 8)\}$. In the second phase, we merge those four communities, shown with thin dashed lines in the middle figure, and obtain the global community G. In the third phase, we attempt merging the global community G with the communities in set GC, but no merge happens. The rightmost figure shows the final global communities. The rightmost table shows the set of local communities merged to form each global community. For example, local communities $2, 7$, and 8 form global community G. Overall, the global community B from the leftmost figure evolved to form the global community G from the rightmost figure.

7.6 SONIC Improvements

In this section, we describe improvements over the SONIC core.

7.6.1 Significant Change Detection

Insertion and removal of edges cause changes in the local community structure of vertices. However, the base version of SONIC does not quantify the *significance* of

these changes. In particular, any change that connects vertices that already have a common neighbor leads to dissolving some global communities and performing the merge step. As an improvement, we introduce the notion of *significant change* with respect to the local community structure of vertices. When there is a change in the local community structure of the *EmEn* of a vertex, we compare the existing community structure to the new community structure using the Normalized Mutual Information (NMI) index [50]. For two groups of local communities, L_1 and L_2, their NMI is defined as follows:

$$\text{NMI}(L_1, L_2) = \frac{I(L_1; L_2)}{(H(L_1) + H(L_2))/2},$$

where $H(L_1)$ and $H(L_2)$ are the entropies of L_1 and L_2, respectively; and $I(L_1; L_2)$ is the mutual information of L_1 and L_2:

$$I(L_1; L_2) = H(L_1, L_2) - H(L_1|L_2) - H(L_2|L_1),$$

where $H(L_1, L_2)$ is the joint entropy of L_1 and L_2, and $H(L_1|L_2)$ is the entropy of L_1 conditional on L_2. The NMI values lie within the range $[0, 1]$, where higher values indicate higher similarity.

We compute the NMI for each vertex whose *EmEn* is impacted (set S in Algorithm 25) by comparing the set of local communities in its *EmEn* before the insertion/removal ($R[u]$ for $u \in S$ in Algorithm 25) with the ones after ($LC(u)$ for $u \in S$ in Algorithm 25 after incremental label propagation). If this similarity is above a specified threshold, then we assume that there is no significant change and do not update

L^- and L^+ with the removed and newly created local communities. Otherwise, we update L^- and L^+ and also set the $R[u]$ to $LC(u)$, i.e., update the local community structure. Continued changes in the graph structure are accumulated if no significant change is observed in the community structure. We denote the threshold by α, and name it as the *significant change threshold*. If $\alpha = 1$, then every local change is accepted as significant, whereas $\alpha = 0$ means that any local change is regarded as insignificant. As such, we take $\alpha \in (0, 1]$

Using the α parameter provides the ability to adjust the trade-off between update cost and the community detection accuracy. This is very useful, especially for scenarios where the update rate of the graph is high relative to the query rate. Setting a lower α means that we do not keep the communities perfectly up to date after each update. If the query rate is low, it is acceptable to have a lower α, as the staleness in the responses will be relatively low compared to the query period. If the query rate is high and the application can tolerate responses with less up-to-date data, it may still be acceptable to have a lower α. This way, less computing resources are spent on edge updates and more resources are available to respond to queries.

7.6.2 Minhash-based merge

In Section 7.5.2, we introduced the incNaive algorithm for performing the re-merge of the communities. However, this algorithm is expected to get costly when the size of the merged communities increase. This is because the cost of computing the overlap similarity is linear in the size of the smaller set, since we keep the communities as

hash sets. As the communities get larger and larger towards the end of the merge process, this cost significantly increases. In this section, we propose an adaptation of the *min-hashing* technique to alleviate this problem.

Min-hashing is a technique for quickly estimating the Jaccard similarity between two sets [32]. Let A and B be two sets, then the Jaccard similarity between them is given by $|A \cap B|/|A \cup B|$. Min-hashing uses n random hash functions to map the elements of the two sets to values, and for each one of the hash functions, it finds the smallest hash values for the two sets. If the smallest hash value for A and B agree for m number of the hash functions, then the Jaccard similarity is estimated as m/n. The probability of the minimum hash values of A and B being the same is equal to the probability of the item having the minimum hash value being in the intersection of the two sets. It is easy to see that the latter is equal to the Jaccard similarity, as there are $|A \cap B|$ items in the intersection and there are $|A \cup B|$ items in total.

The speed advantage of min-hashing compared to the explicit computation is that, min-hashing based similarity can be computed in $\mathcal{O}(n)$ time, where n is the number of hash functions used. This number is expected to be smaller than the size of the sets. Importantly, we assume that the min-hashes are computed once for all the sets and many comparisons are made over these sets to compute pairwise Jaccard similarities. The min-hashing based computation of the similarity will lose its accuracy if the number of hash functions is small. As a result, there is a trade-off between performance and accuracy that can be adjusted by setting n properly.

One important problem in using min-hashing for our merge problem is that min-hashing is based on Jaccard similarity, whereas we use overlap similarity for our merge process. To convert a given overlap similarity coefficient (β) to the corresponding Jaccard similarity coefficient (θ), we apply the following formula:

$$\theta = \frac{\beta}{(1-\beta) + |B|/|A|},$$

where $|A| \leq |B|$. This is obtained as follows. We have $\beta = |A \cap B|/|A|$ from the definition of overlap similarity. Thus, $|A \cap B| = \beta \cdot |A|$. Since $|A \cup B| = |A| + |B| - |A \cap B|$, by plugging in $\beta \cdot |A|$ in place of $|A \cap B|$, we get $|A \cup B| = (1-\beta) \cdot |A| + |B|$. From the definition of Jaccard similarity we have $\theta = |A \cap B|/|A \cup B|$ and plugging in our derivations of $|A \cap B| = \beta \cdot |A|$ and $|A \cup B| = (1-\beta) \cdot |A| + |B|$, we get $\theta = \beta/((1-\beta) + |B|/|A|)$.

In our min-hash based merge, we compute the n min-hash values for each one of the local communities to be merged only once. Later, when we merge two communities, we only need $\mathcal{O}(n)$ operations to compute the new min-hash values for the merged community, as we only need to take the smaller of the min-hash value pairs for each hash function. As a result, we perform hashing only once over the base communities and re-use the results many times during the merge. We evaluate the accuracy vs. performance trade-off involved in setting the number of hash functions as part of our experimental evaluation in Section 7.7.

7.6.3 Inverted-Index based merge

In Section 7.6.2, we proposed the use of min-hashing as a cheaper alternative to the explicit overlap similarity computations performed within the INCNAIVE algorithm during the re-merge of the communities. While min-hashing reduces the cost of similarity computations, especially for large communities, the number of such computations made for comparing communities for possible merges is still high. One idea that comes to mind to alleviate this problem is *locality-sensitive hashing* (*LSH*) [89]. The motivation behind using locality-sensitive hashing is to hash similar items to the same buckets, so that the pair-wise similarity comparisons can be limited to the confines of individual buckets. For our re-merge problem, this would significantly reduce the number of similarity computations made and thus the overall merge time. However, it has been shown that locality sensitive hash functions do not exist for the overlap similarity metric [41]. As a result, the LSH technique cannot be adapted for our problem.

In this section, we propose to leverage our support data structures, particularly the global community ids for each vertex, to reduce the number of comparisons made during the re-merge. This alternative merge algorithm, called INCINVINDEX, is based on inverted indices of global communities. The algorithm attempts to make small number of comparisons by traversing the vertices of communities to be merged and computing their intersections with the surrounding communities via a simple counting procedure, utilizing fast lookups and updates on a map structure.

Algorithm 27 gives the pseudocode of INCINVINDEX. The algorithm consists of two phases. The first one is called the *Pre-formation Phase* (lines 1 to 3). In this phase, we collect both the local communities within the dissolved global communities ($F \in \mathcal{F}$) and the newly created local communities (L^+) into L, called the *global merge list* (line 1). We then update the data structure that keeps the list of global communities each vertex belongs to (line 3). This data structure serves as the inverted index. For each vertex u in a local community l within the global merge list L, we remove the dissolved communities (GC^C in the pseudocode, denoting anything other than the intact communities) from its list of global communities $GC(u)$, and add the local community l as a global community to $GC(u)$. After the first step is complete, we are ready to merge L with the intact global communities in GC. Note that we perform the update of the inverted index $GC(u)$ as part of the first phase, so that we can do efficient merges in the second phase.

The second phase is called the *Global Merge Phase* (lines 4 to 20). In this phase, we merge L and GC. For each community $l \in L$, we check to see whether it can merge with any other community in L or GC. But rather than doing this by iterating over L and GC, we do it by using the inverted index. In particular, for each community $l \in L$, we iterate over its vertices. For each vertex $u \in l$, we go over the global communities that contain it. These communities are listed in the inverted index, as $G(u)$. For each such community $g \in GC(u)$, we increment a counter stored in a map data structure, denoted by M (line 9). $M[g]$ represents the current count of common

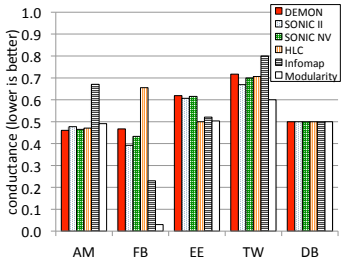


Figure 7.2: Conductance on real-world graphs. Modularity is the best, as it is an optimization algorithm for conductance.

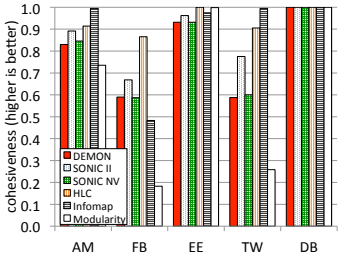


Figure 7.3: Cohesiveness on real-world graphs. Results depend on the graphs.

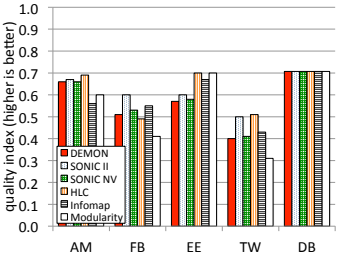


Figure 7.4: Quality index scores on real-world graphs. DEMON and SONIC variants show competitive behavior.

vertices between l and g. The moment this value is high enough to satisfy the overlap similarity condition (line 10), we can merge l and g. We do this by merging g into l. We then remove g from either GC or L, depending on which one it came from. Finally, we update the inverted index by removing g from the list of global communities $G(v)$ of each vertex $v \in l$, and adding l into the same (see line 17). Once a merge happens, we need to break and go back to the start of processing l for new merges (line 19). For this purpose, a boolean variable c is kept to break out of the inner two for loops at once.

Note that, as in the INCNAIVE algorithm, the outer loop only goes over the global merge list L. As before, we know that once no more merges can happen between L and any other community in L or GC, the intact communities in GC cannot get

involved in any merges, since they cannot merge among themselves. As a result, we return the final set of global communities as $L \cup GC$ (line 20).

7.7 Experimental Evaluation

This section presents the evaluation of our algorithms using various datasets under different scenarios. The first set of experiments focus on comparing the proposed algorithms to the previous work with respect to the quality of the identified communities. The second set evaluates the running time performance of our algorithms when processing real-world datasets of different types and sizes. The third set compares the running time performance of the different merge algorithms introduced in Sections 7.5.2, 7.6.2 and 7.6.3. The fourth set investigates the impact of the two algorithmic parameters, namely the significant change threshold (α) and the merge factor (β), on the algorithm's running time performance and community detection quality. The last set of experiments compare the running time performance of our algorithms when processing synthetic graphs of different sizes.

Setup. Algorithms were implemented in C++ and compiled with `gcc 4.8.1` at -O3 optimization level. Experiments were executed sequentially on a Linux operating system running on a machine with an Intel Xeon E5520 2.27GHz CPU and 48GBs of RAM.

Datasets. We obtained real-world datasets from SNAP [166]. They are the co-purchasing network (amazon0601 (AM)), friendship network (facebook (FB)), follower-followee network (twitter (TW)) and email communication network (email-Enron (EE)). We also extracted the co-authorship network of DBLP papers[6]. Table 7.1 shows the properties of these datasets. For each graph, we give its size, the time taken to run the non-incremental find-and-merge algorithm and the memory space overhead (in terms of number edges) of the support data structures used by our algorithms. We also use synthetic graphs in our experiments, in order to better evaluate the impact of changing graph size. These graphs, generated using SNAP's R-MAT generator [166], follow a power law vertex degree distribution and exhibit small world properties. To achieve that, we set the partition probabilities of the generator to $[0.40; 0.25; 0.20; 0.15]$. We set the average degree of the R-MAT graphs to 4.

7.7.1 Quality

In this section we evaluate the quality of the core SONIC algorithm and the improvements introduced in Section 7.6, using real-world datasets. We use four previously published state-of-the-art community detection algorithms for comparative evaluation: Hierarchical Link Clustering (HLC) [5], Infomap [141], Modularity [130], and DEMON [47]. HLC has been shown to outperform other existing overlapping community detection algorithms. Infomap is a non-overlapping algorithm that aims to minimize the random walk entropy. Modularity is an eigenvector-based non-overlapping

[6] www.informatik.uni-trier.de/ ley/db/

community detection method, which maximizes the modularity metric. DEMON serves as our baseline, since it is the non-incremental (static) version of SONIC. It is worth noting that our goal in this comparison is twofold: showing that (*i*) SONIC performs similar to DEMON, (*ii*) SONIC is a competitive community detection algorithm in terms of quality.

For the SONIC algorithms, we construct the communities by first bootstrapping them with DEMON for the entire graph, then remove the randomly selected 10% of edges by applying SONIC at each step and then inserting the same 10% by again applying SONIC at each step. This way, we capture the impact of SONIC on the quality of the communities. The significant change threshold α and the merge factor β are both set to 1.0 to provide up-to-date and deterministic results. Remember that, if β value is less than 1.0, output is non-deterministic (Theorem 13). In all figures, SONIC NV represents core SONIC using incNaive merge algorithm (Section 7.5.2), SONIC II represents SONIC using the incInvIndex merge algorithm (Section 7.6.3) and SONIC MHx is SONIC using the minhash-based merge algorithm (Section 7.6.2) with x number of hash functions.

In the first experiment, we quantify the quality of the communities found using two metrics: *conductance* and *cohesiveness*. The conductance metric measures how connected a community is to the rest of the graph. It measures the fraction of total edge volume pointing outside of the community. If we denote the number of edges crossing the boundaries of the community as $|E_o|$ and the total number of

edges of the community as $|E|$, then conductance is given by $c_d = |E_o|/|E|$. Lower conductance values imply better communities. In networks that contain overlapping communities, communities are not disjoint and thus conductance is expected to be relatively high compared to those that contain non-overlapping communities. The cohesiveness metric quantifies how connected the members of a community are to each other. That is, it measures the density of a community. If we denote the number of edges within the boundaries of the community as $|E_i|$ and the total number of vertices in the community as $|V|$, then cohesiveness is given by $c_h = |E_i|/(|V| \cdot (|V|-1|)/2)$. Higher values imply better communities. We also define a *quality index* by combining conductance and cohesiveness by taking their geometric mean, that is $q = \sqrt{(1-c_d) \cdot c_h}$. The quality index metric provides a bigger picture to show the impact of both conductance and cohesiveness. Algorithms balancing the two metrics are expected to give higher scores for *quality index*. Since the number of communities reported by each competitor algorithm is significantly different, we focus on top $1,000$ communities with the best quality index score and report the geometric means.

Figure 7.2 shows the results for the conductance metric. Modularity is the best performing algorithm for most graphs since it optimizes the modularity metric to find non-overlapping communities. The modularity metric measures the fraction of edges that fall within the given communities minus the expected fraction of such edges if all edges were distributed at random. Thus, it is closely related with the conductance metric. We observe that DEMON, SONIC NV, and SONIC II perform very similar to

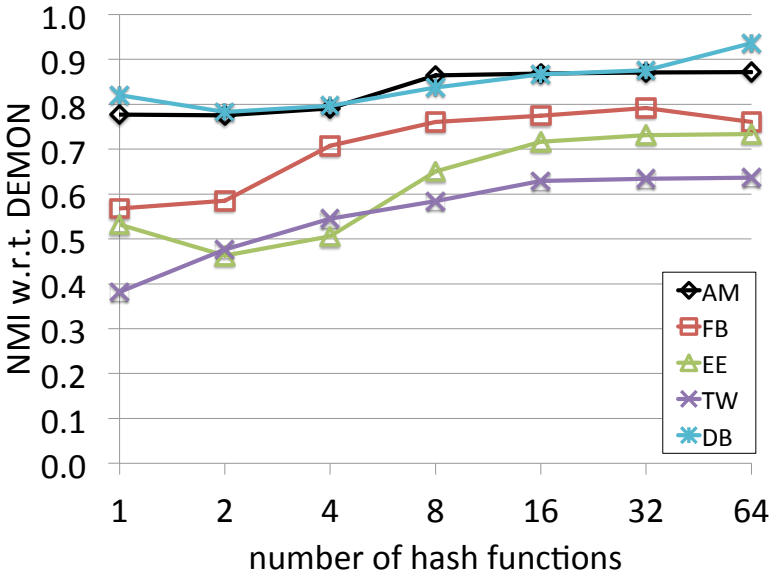


Figure 7.5: NMI scores of SONIC MH wrt. DEMON with varying # of hash functions on real-world graphs.

each other and rank competitive. For DBLP_coauthor graph, all algorithms report the same 1,000 communities as the best ones, so their quality scores are same. Figure 7.3 shows the cohesiveness results. HLC and Infomap perform best for this metric in all graphs and again DEMON, SONIC NV, and SONIC II perform similar to each other and rank well.

Figure 7.4 shows the quality indexes for all graphs. SONIC variants and DEMON show similar results and give almost the best results on amazon0601 and facebook graphs. Overall, SONIC and DEMON provide a *good balance* between cohesiveness and conductance, which cannot be said for any of the other algorithms.

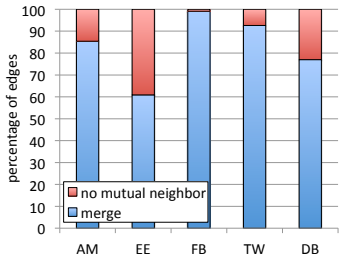


Figure 7.6: Most edge removal/insertions result in a merge. Yet, for some graphs, a sizable fraction of updates skip the merge.

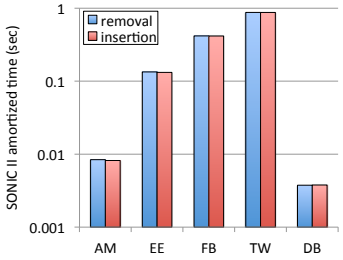


Figure 7.7: Amortized runtimes of one edge removal and insertion on real-world graphs when $1,000$ edges are removed and inserted.

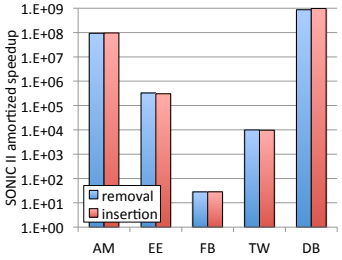


Figure 7.8: Amortized speedup of one edge insertion/removal w.r.t. static algorithm when $1,000$ edges are removed/inserted.

We also investigate how the number of hash functions affect the quality of the communities found by SONIC when using the minhash-based merge. For this purpose, we obtained the communities with different number of hash functions (from 1 to 64) and measured the similarity of the results to those obtained by running the DEMON algorithm. We applied a version of Normalized Mutual Index (NMI) that is adopted for overlapping communities [103] as a scoring function to determine the similarity between two sets of communities computed by SONIC and DEMON. If the two sets of community are identical, then their NMI score is a perfect 1. Figure 7.5 shows the NMI scores of SONIC with respect to DEMON algorithm. As expected, increasing the number of hash functions provides results that are more similar to DEMON, since the NMI increases. The same trend is observed for all graphs. For most of the graphs, the NMI stabilizes after 16 hash functions.

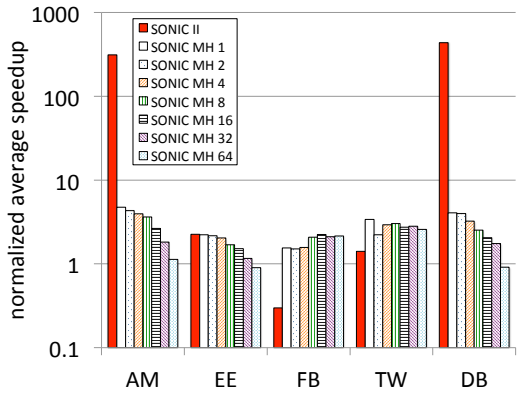


Figure 7.9: Normalized insertion/removal speedups of SONIC variants w.r.t. SONIC NV. SONIC II performs best for large networks.

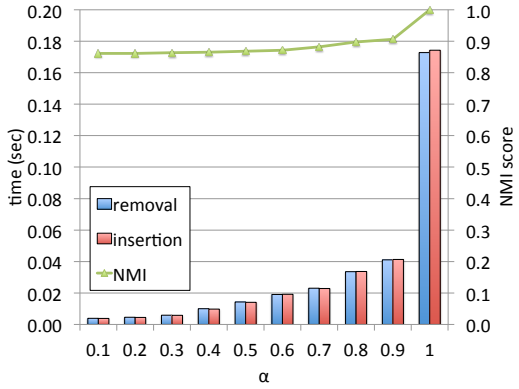


Figure 7.10: Impact of α on the email-Enron dataset. Lower values of α provide significant speedups with little impact on quality.

7.7.2 Running Time Performance

In this section, we evaluate the running time performance of SONIC II, which is expected to be the best algorithm, on real-world graphs. Since all of our graphs are originally static, we emulate the streaming algorithms by considering that the whole set of vertices and edges constitute a *sliding window* snapshot. To evaluate streaming execution, we first evict a random edge from the current graph. This emulates the behavior of a full sliding window which opens space for inserting a new edge. We then insert the *same edge* to preserve the graph structure. In this way, we preserve the structure of the real dataset. As an important note, we do not assume any specific data distribution with respect to which edges get inserted or removed. Furthermore,

we make no assumptions regarding edge arrival rates. Instead, we use our algorithms to process repeated removals and insertions as fast as possible.

We measure the average execution time for removing and inserting one edge to each dataset. We set the significant change threshold α and the merge factor β to 1.0. For speedup results, we compute the speedup of each single edge update with respect to non-incremental (static) community construction and report the geometric mean of speedups over multiple updates. Figure 7.6 shows the relative frequency of the two code paths executed by SONIC (Algorithm 25) when inserting/removing the $1,000$ randomly picked edges, for each dataset. Recall that when the vertices connected by the edge have *no mutual neighbors*, then the algorithm terminates early. If they share neighbors, then the algorithm executes the merge step. We observe that, in general, most updates result in a merge. However, depending on the structure of the graph, a non-significant number of edges may take the early termination path. For the facebook and twitter graphs, more than 93% of the edges result in a merge operation, whereas this number is 85%, 60% and 77% on amazon0601, email-Enron and DBLP_coauthor graphs, respectively. The reason is that facebook and twitter graphs have higher average degree and thus are denser than other graphs. When the graph is denser, the probability of having a mutual neighbor for the incident vertices of a randomly selected edge is higher.

Figure 7.7 shows the amortized running times of a single edge insertion and a single edge removal. For all graphs, we manage to stay below 1 sec per edge insertion

or removal. For the biggest two graphs, amazon0601 and DBLP_coauthor, SONIC II performs better, keeping the insertion/removal time below 0.01 sec.

Figure 7.8 shows the speedup of a single edge insertion and a single edge removal relative to the non-incremental find-and-merge algorithms. We compute the speedup by dividing the from-scratch computation time of entire graph to the one edge insertion/removal time. We compute the geometric mean of all speedups. The resulting speedup increases as the graph size gets larger, since it takes more time to re-compute communities from scratch. For the amazon0601 and DBLP_coauthor graphs, 8 and 9 orders of magnitude speedups are reached, respectively, which is impressive.

Experiments on Real Temporal Data:

Apart from the sliding window emulation scenario, we also investigated the performance of the SONIC II algorithm using real temporal data from the DBLP_coauthor graph, which has an explicit ordering on the stream of edges based on timestamps. In particular, we inserted the co-authorship edges for the papers published after January 1, 2013 and measured the resulting execution time and speedups. Average execution time is 0.018 sec per edge insertion and average speedup observed is 405M, which is 8 orders of magnitude speedup over from-scratch computation.

7.7.3 Comparison of Merge Variants

In this section, we compare the runtime performance of different merge algorithms, namely SONIC NV, SONIC II, and the SONIC MH variants. We use the same experiment setup as in Section 7.7.2.

Figure 7.9 shows the average of normalized insertion and removal speedups of SONIC II and SONIC MH variants with respect to SONIC NV. The best performing merge algorithm depends on the dataset. For the amazon0601 and DBLP_coauthor graphs, SONIC II performs the best with a significant difference, as it is 312 and 435 times faster than SONIC NV, respectively. In general, SONIC MH variants perform better as the number of hash functions decrease. SONIC MH1 is 2.95 times faster than SONIC NV whereas SONIC MH64 is 1.38 times faster. Considering the large sizes of the amazon0601 and DBLP_coauthor graphs, the low runtime of SONIC II can be explained by the efficient merge operations. SONIC II, as explained in Section 7.6.3, tries to merge only the spatially close communities (that have common vertices), therefore provides an efficient merge operation. For the SONIC MH variants, the trend is expected because as the number of hash functions decrease, the size of the merged community signatures decrease as well, which results in lower execution times.

The email-Enron graph shows trend similar to the amazon0601 and DBLP_coauthor graphs for the SONIC MH variants. However, SONIC II is only slightly better than the best MH variant. The facebook graph shows a different trend, where SONIC NV is the fastest option.

Summary. Overall, the size and the structure of the graph have a significant impact on the merge variant to be selected. For large size networks, like amazon0601, SONIC II is the best option, whereas, denser graphs with smaller sizes, like facebook

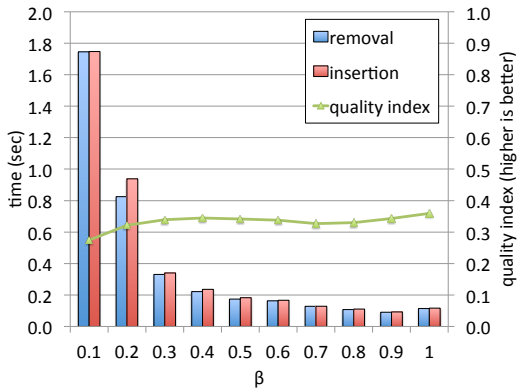


Figure 7.11: Impact of β on the average execution time of insertions and removals. Runtimes get slower with lower values of β. Quality index does not significantly change when varying β.

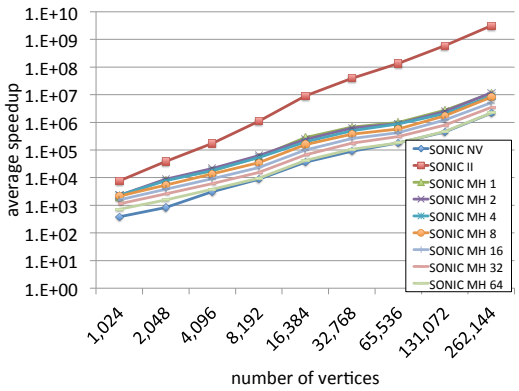


Figure 7.12: Average of removal and insertion speedups on R-MAT graphs as a function of the graph size. All merge variants show increasing speedups with increasing scale. SONIC II has the best scalability, reaching $3.1B$ times speedup.

and twitter, SONIC MH variants can give better performance with best fitting number of hash functions. However, we need to keep in mind that with very few hash functions, the quality results of the SONIC MH variants are not good (Figure 7.5). As such, we conclude that the SONIC II merge algorithm is the most robust option for general use.

7.7.4 The α and β Effect

In this section, we report the impacts of the significant change threshold (α) and the merge factor (β) on the running time performance and community detection quality of SONIC.

For the significant change threshold experiment, we selected the email-Enron dataset. We removed and inserted $1,000$ edges to our dataset using the SONIC NV algorithm and experimented with different α values from 0.1 to 1.0. The β value is fixed at 1.0. Remember that as the α value decreases, changes in the local community structure are regarded as less significant and, therefore, less merge operations occur. After the removals and insertions, we compute the NMI score of the communities resulting from an α value variant with respect to the communities resulting from a setting of $\alpha = 1.0$. This way, we can see how much divergence occurs due to the lower α values.

Figure 7.10 shows the NMI scores (using the right y-axis) for each α variant. The figure also shows the average time taken (on the left y-axis) for removing and inserting an edge as α varies. When we set α to 0.1, the quality degradation is not that significant, as the average NMI decreases by only 14%. On the other hand, the removal/insertion of an edge executes 45 times faster. Another observation is that even if we set α to 0.9, we have speedups of 10.7 times, while sacrificing little in quality (10%). These results show the advantage of using lower values for α parameter.

For the merge factor experiment, we chose the facebook dataset. As before, we used the SONIC NV algorithm, but with a fixed value of α (1.0) and for different values of β (0.1 to 1). Figure 7.11 shows the running time (using the left y-axis) and the quality index (on the right y-axis) introduced in Section 7.7.1. We observe slower running times as β decreases. The reason is that, increased number of merges are

happening with lower values of β. On the other hand, the quality index for different β values show a little variability and the quality of the communities resulting from the settings we have used in the experiments, i.e., $\beta = 1.0$, is decent.

7.7.5 Scalability

In this section, we report the scalability of SONIC and its variants when processing the synthetic R-MAT graphs of different sizes, which vary from 2^{10} to 2^{18} vertices. We set both α and β to 1.0.

Figure 7.12 shows the average speedup for a single edge removal and insertion as a function of increasing R-MAT graph size. We compute speedups with respect to the from-scratch computation of communities with the DEMON algorithm. All of the proposed algorithms present a scalable behavior with the increasing graph size. As the scale gets larger, SONIC II shows outstanding performance, reaching to 3.1B times speedup, which is 9 orders of magnitude speedup over the from-scratch computation. SONIC NV and SONIC MH variants show decent scalability results as well, reaching 6 and 7 orders of magnitude speedups, respectively, but they are not better than SONIC II. Considering the quality being traded off by SONIC MH variants with small number of hashes, SONIC II turns out to be the best performing algorithm for our scalability experiments.

The scalability experiments indicate how good our streaming algorithms can perform for different graph sizes when there are *read queries* (asking communities of a vertex) interspersed with *write queries* (insertion and removal operations). Taking

the R-MAT graph with the size of 2^{18}, we can see that if write/read ratio is less than $3.1B$ (the average speedup of one removal and one insertion), it is better to use our SONIC II algorithm than computing communities from scratch.

7.8 Summary

In this chapter, we introduced incremental algorithms for the streaming overlapping community detection problem. The main benefit of these algorithms is that they provide maintenance of global community ids of the vertices in the graph, as updates are taking place. This avoids the from-scratch computation of communities and brings the ability to serve fresh community results to on-demand queries.

Our core SONIC algorithm produces high quality communities and is efficient in terms of running time, when applied on the real-world and synthetic graphs of different sizes. The core SONIC algorithm is further enhanced by techniques such as the significant change detection, minhash based merge, and inverted-index based merge. For instance, we reach 9 orders of magnitude speedup with our SONIC variant that uses inverted-index based merge, compared to the from-scratch (non-incremental) alternatives, on synthetic R-MAT graphs of size 2^{18}. Given the dynamic nature of social networks and the importance of community detection analytics, we believe that our incremental algorithms will be beneficial in many real-world applications with streaming update requirements. Thanks to the outstanding execution times, our algorithms will make it possible to analyze complete dynamic scenarios so that community evolution trends can be observed in real social networks.

Algorithm 26: incNaive Merge$(G, \beta, GC, \mathcal{F}, L^+)$

Data: G: graph, β: merge factor, GC: global communities, $\mathcal{F}$: local community sets of previously dissolved global communities, L^+: newly created local communities

▷ Perform the Distribute New Phase

for each $l \in L^+$ **do** ▷ For each new local comm.
  ▷ Find a suitable dissolved comm. for l
  Find an F s.t. $\exists f \in F, f \cap l \neq \emptyset$
  $F \leftarrow F \cup \{l\}$ ▷ Add local comm. to a dissolved comm.

▷ Perform the Regroup Dissolved Phase

$L \leftarrow \emptyset$ ▷ Initialize the global merge list
for each $F \in \mathcal{F}$ **do** ▷ For each dissolved global comm.
  for each $f_i \in F$ **do**
    for each $f_j \in F$, where $j > i$ **do**
      if overlap$(f_i, f_j) \geq \beta$ **then** ▷ There is a merge
        $f_i \leftarrow f_i \cup f_j$ ▷ Merge latter into former
        $F \leftarrow F \setminus f_j$ ▷ Get rid of the latter
  $L \leftarrow L \cup F$ ▷ Add merged comms. to global merge list

▷ Perform the Global Merge Phase

for each $c_i \in L$ **do**
  for each $c_j \in L$, where $j > i$ **do**
    if overlap$(c_i, c_j) \geq \beta$ **then** ▷ Meets merge criteria
      $c_i \leftarrow c_i \cup c_j$ ▷ Merge latter into former
      $L \leftarrow L \setminus c_j$ ▷ Get rid of the latter
  for each $g_k \in GC$ **do**
    if overlap$(c_i, g_k) \geq \beta$ **then** ▷ Meets merge criteria
      $c_i \leftarrow c_i \cup g_k$ ▷ Merge latter into former
      $GC \leftarrow GC \setminus g_k$ ▷ Get rid of the latter
$GC \leftarrow L \cup GC$ ▷ Update the global comms.

Algorithm 27: incInvIndex Merge$(G, \beta, GC, \mathcal{F}, L^+)$

Data: G: graph, β: merge factor, GC: global communities, $\mathcal{F}$: local community sets of previously dissolved global communities, L^+: newly created local communities

▷ Perform the Pre-Formation Phase

$L \leftarrow \cup_{F \in \mathcal{F}} F \cup L^+$ ▷ Put local comms. into the merge list

for $l \in L$ **do** ▷ For each comm. in the global merge list

  ▷ Update the global community mappings

  $GC(v) \leftarrow GC(v) \setminus GC^C \cup \{l\}, \forall_{v \in l}$

▷ Perform the Global Merge Phase

for each $l \in L$ **do** ▷ For each comm. in the merge list

  $M \leftarrow \{\}_0$ ▷ Initialize the intersection counter map

  $c \leftarrow$ **false** ▷ Initialize the change flag (no changes)

  for each $u \in l$ **do** ▷ For each vertex in the comm.

    for each $g \in GC(u)$ **do** ▷ For global comms. of u

      $M[g] \leftarrow M[g] + 1$ ▷ Incr. # of intersections

      ▷ Already meets merge criteria

      if $(M[g]/min(|l|, |g|)) \geq \beta$ **then**

        $c \leftarrow$ **true** ▷ Mark the change

        $l \leftarrow l \cup g$ ▷ Merge the communities

        if $g \in GC$ **then** ▷ g is from intact comms.

          $GC \leftarrow GC \setminus g$ ▷ Remove g from GC

        else ▷ g is from the global merge list

          $L \leftarrow L \setminus g$ ▷ Remove g from L

        ▷ Update the global community mappings

        $GC(v) \leftarrow GC(v) \setminus \{g\} \cup \{l\}, \forall_{v \in l}$

        break ▷ Break out of the loop

    if c *is* **true then break** ▷ Re-merge the new l

$GC \leftarrow GC \cup L$ ▷ Update the global comms.

Graph dataset	Number of vertices	Number of edges	Average degree	Batch time	Memory overhead
amazon0601 (AM)	403,394	3,387,388	8.39	16h	$4.25 \cdot \|E\|$
facebook (FB)	4,039	88,234	21.84	5.530s	$4.17 \cdot \|E\|$
email-Enron (EE)	36,692	367,662	10.02	3.42m	$3.59 \cdot \|E\|$
twitter (TW)	81,306	2,684,324	33.01	21.53m	$3.68 \cdot \|E\|$
DBLP_coauthor (DB)	1,236,220	15,897,220	12.85	76h	$4.79 \cdot \|E\|$

Table 7.1: Real-world graph datasets and their properties

Chapter 8: Conclusion, Future Plans and Open Problems

In this study, we proposed fast algorithms for different large-scale network analytic problems. Firstly, we investigated how to manipulate the large networks by leveraging special structures in the graph for fast centrality computation in Chapter 2. **BADIOS** is introduced in that respect for fast betweenness and closeness centrality computation. Then, we presented solutions for fast centrality computation on cutting-edge hardware in Chapter 3. We made use of different techniques, such as simultaneous BFS application for betweenness and closeness centrality computation, and vectorization for closeness centrality computation. After that, we proposed different incremental/streaming algorithms for sliding window scenarios on different problems. The problem of closeness centrality computation for dynamic graphs is investigated and efficient computation filtering algorithms are presented in Chapter 4. Furthermore, these algorithms are parallelized using a parallel framework, Streamer. Streaming algorithms for dense subgraph discovery (k-core decomposition) and community detection problems are also investigated and efficient solutions are proposed in Chapters 5 and 7. Obtained speedups are impressive and show the significancy of our solutions for large-scale network analytics. Last, but not least,

new dense subgraph discovery algorithms are introduced to find high-quality dense subgraphs with hierarchy in Chapter 6, which are superior than state-of-the-art techniques.

We hope that this dissertation will be useful for computer science people and domain scientists working in sociology, bioinformatics, and web science. Each graph analytics we have studied have many applications in those domains, and our algorithms can be leveraged for making sense of different type of networks in a more efficient manner under different setups, like incremental and parallel scenarios.

8.1 Limitations

There are some challenges we have faced in our work, which indicates some limitations on our studies. Firstly, all of our algorithms are exact, in the sense that we do not approximate a graph analytic. This may not be realistic for real-world use cases, but we believe that our contributions can be used as a building block for further analyses. Secondly, evaluating the results of a new graph analytics based on the ground-truth information is quite challenging. For the dense subgraph discovery and community detection problems, existing ground-truth information is quite dependent to the domain. For instance, the densest region in a protein-protein interaction network may not correspond to something meaningful and larger subgraphs with lower densities are more of interest. Obtaining the "true" ground-truth for all domains, and matching those ground-truth information with a generic algorithm does not seem to

be a realistic goal. Instead, we believe that different algorithms might be adapted for different domains while benefiting from the generic algorithms proposed.

8.2 Future Plans

We have several future directions for each of the work in this study. Our future plans, classified with respect to the graph analytics problems, are as follows.

8.2.1 Fast and Incremental Centrality Computation

For this work, we plan to investigate incremental batch algorithms, where a set of edges are inserted/removed from the graph at once. In a real-life scenario, write and read queries do not have equal frequency. Most of the time, write queries are much more frequent than the read queries. Accumulating the changes on the graph and answering the read queries by minimum amount of work requires batch algorithms, where a set of edges are inserted/removed to graph. In that respect, we plan to work on batch algorithms for incremental closeness centrality computation, which will result in more efficient solutions than the single edge insertion/removal algorithms.

8.2.2 Incremental and High-Quality Dense Subgraph Discovery

The ongoing research for this work is pull-based scheduling algorithms for incremental settings where algorithm is graph-oblivious. The assumptions in [145] is for push-based settings, where each change in the topology of network is immediately

known by the algorithm designer. However, in the pull-based method, this assumption is no more valid. Graph traversal is assumed to be expensive and each topological change can only be learned by probing a specified vertex. The goal is to find the set of vertices to be probed so that the most accurate k-core decomposition is maintained for the entire graph. A similar setting is used for page-rank problem in [18].

Apart from that, we plan to investigate application areas where our nucleus decomposition is useful. One ongoing work is finding the dense subgraphs and hierarchy among them in protein gene interaction networks. We plan to identify critical dense regions in these networks and collaborate with domain scientists to make sense of these subgraphs.

8.2.3 Streaming Overlapping Community Detection

Possible future works for this project includes the incremental batch algorithms, and parallelization on cutting-edge hardware. Apart from the fast algorithms for this problem, we also plan to investigate more meaningful metrics designed for overlapping community detection problem.

8.3 Open Problems

Apart from our future plans, there are some interesting open problems for researchers to extend our work.

For graph manipulation via shattering and compressing the networks, new special structures can be investigated. One such special subgraph is the two connected vertices, where each vertex has degree of two. The interesting thing about this structure is that, betweenness centrality and closeness centrality scores of those two vertices are equal, since they are isomorphic. This equality can be leveraged and graph can be compressed by replacing those two vertices and the edge between them with a super vertex. The open problem with this approach is coming up with an attribute to be maintained for correct centrality computation on the compressed graph.

Apart from the algorithmic solutions, a parallel incremental graph processing framework can be designed and implemented for a set of graph problems. One common property in streaming k-core decomposition and streaming overlapping community detection problems is that an impacted subgraph needs to be located upon each edge insertion/removal. Parallelizing the set of insertions/removals without touching their associated subgraphs is an interesting problem. Different scheduling techniques can be investigated in that respect. Furthermore, an optimistic approach can be chosen with a possible roll-back strategies for the accurate maintenance of graph analytics.

Bibliography

[1] University of florida sparse matrix collection. http://www.cise.ufl.edu/research/sparse/matrices/.

[2] M. K. Agarwal, K. Ramamritham, and M. Bhide. Real time discovery of dense clusters in highly dynamic graphs: Identifying real world events in highly dynamic environments. *Proc. VLDB Endow.*, 5(10), June 2012.

[3] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader. Scalable graph exploration on multicore processors. In *SuperComputing*, pages 1–11, 2010.

[4] C. Aggarwal and K. Subbian. Evolutionary network analysis: A survey. *ACM Comput. Surv.*, 47(1):10:1–10:36, May 2014.

[5] Y.-Y. Ahn, J. P. Bagrow, and S. Lehmann. Link communities reveal multiscale complexity in networks. *Nature*, 466(7307):761–764, 2010.

[6] H. Aksu, M. Canim, Y. Chang, I. Korpeoglu, and O. Ulusoy. Distributed k-core view materialization and maintenance for large dynamic graphs. *Knowledge and Data Engineering, IEEE Transactions on*, PP(99):1–1, 2014.

[7] J. I. Alvarez-Hamelin, A. Barrat, and A. Vespignani. Large scale networks fingerprinting and visualization using the k-core decomposition. In *Advances in Neural Information Processing Systems 18*, pages 41–50, 2006.

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

[9] R. Andersen and K. Chellapilla. Finding dense subgraphs with size bounds. In *Workshop on Algorithms and Models for the Web Graph (WAW)*, pages 25–37, 2009.

[10] R. Andersen and K. Chellapilla. Finding dense subgraphs with size bounds. In *Workshop on Algorithms and Models for the Web-Graph (WAW)*, pages 25–37, 2009.

[11] A. Angel, N. Sarkas, N. Koudas, and D. Srivastava. Dense subgraph maintenance under streaming edge weight updates for real-time story identification. *Proc. VLDB Endow.*, 5(6):574–585, Feb. 2012.

[12] A. Angel, N. Sarkas, N. Koudas, and D. Srivastava. Dense subgraph maintenance under streaming edge weight updates for real-time story identification. *Proc. VLDB Endow.*, 5(6), Feb. 2012.

[13] Y. Asahiro, K. Iwama, H. Tamaki, and T. Tokuyama. Greedily finding a dense subgraph. *J. Algorithms*, 34(2):203–221, Feb. 2000.

[14] M. Ashburner, C. A. Ball, J. A. Blake, D. Botstein, H. Butler, J. M. Cherry, A. P. Davis, K. Dolinski, S. S. Dwight, J. T. Eppig, M. A. Harris, D. P. Hill, L. Issel-Tarver, A. Kasarskis, S. Lewis, J. C. Matese, J. E. Richardson, M. Ringwald, G. M. Rubin, and G. Sherlock. Gene ontology: tool for the unification of biology. The Gene Ontology Consortium. *Nature genetics*, 25(1):25–29, May 2000.

[15] S. Asur, S. Parthasarathy, and D. Ucar. An event-based framework for characterizing the evolutionary behavior of interaction graphs. *ACM Trans. Knowl. Discov. Data*, 3(4):16:1–16:36, Dec. 2009.

[16] G. D. Bader and C. W. V. Hogue. An automated method for finding molecular complexes in large protein interaction networks. *BMC Bioinformatics*, 4, 2003.

[17] M. Baglioni, F. Geraci, M. Pellegrini, and E. Lastres. Fast exact computation of betweenness centrality in social networks. In *ASONAM*, 2012.

[18] B. Bahmani, R. Kumar, M. Mahdian, and E. Upfal. Pagerank on an evolving graph. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '12, pages 24–32, New York, NY, USA, 2012. ACM.

[19] B. Balasundaram, S. Butenko, and I. Hicks. Clique relaxations in social network analysis: The maximum k-plex problem. *Operations Research*, 59:133–142, 2011.

[20] A.-L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

[21] H. Barcelo, X. Kramer, R. Laubenbacher, and C. Weaver. Foundations of a connectivity theory for simplicial complexes. *Advances in Applied Mathematics*, 26(2):97 – 128, 2001.

[22] V. Batagelj and M. Zaversnik. An O(m) algorithm for cores decomposition of networks. *The Computing Research Repository (CoRR)*, cs.DS/0310049, 2003.

[23] M. Baur, M. Gaertler, R. Görke, M. Krug, and D. Wagner. Augmenting k-core generation with preferential attachment. *Networks and Heterogeneous Media*, 3(2):277–294, 2008.

[24] D. J. Beal, R. Cohen, M. J. Burke, and C. L. McLendon. Cohesion and performance in groups: A meta-analytic clarification of construct relation. *Journal of Applied Psychology*, 88:989–1004, 2003.

[25] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. In *Proceedings of Supercomputing (SC)*, 2012.

[26] M. Belgin, G. Back, and C. J. Ribbens. Pattern-based sparse matrix representation for memory-efficient SMVM kernels. In *Proc. of ICS*, pages 100–109, 2009.

[27] M. D. Beynon, T. Kurç, Ü. V. Çatalyürek, C. Chang, A. Sussman, and J. Saltz. Distributed processing of very large datasets with DataCutter. *Parallel Computing*, 27(11):1457–1478, Oct. 2001.

[28] C. C. Bilgin and B. Yener. Dynamic network evolution: Models, clustering, anomaly detection.

[29] U. Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25(2), 2001.

[30] U. Brandes. On variants of shortest-path betweenness centrality and their generic computation. *Social Networks*, 30(2), 2008.

[31] U. Brandes and C. Pich. Centrality estimation in large networks. *I. J. Bifurcation and Chaos*, 17(7), 2007.

[32] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-wise independent permutations. *Journal of Computer and System Sciences*, 60, 1998.

[33] C. Bron and J. Kerbosch. Algorithm 457: Finding all cliques of an undirected graph. *Commun. ACM*, 16(9):575–577, Sept. 1973.

[34] G. Buehrer and K. Chellapilla. A scalable pattern mining approach to web graph compression with communities. In *Proc. of the 2008 International Conference on Web Search and Data Mining*, WSDM '08, pages 95–106, 2008.

[35] A. Buluç and J. R. Gilbert. The combinatorial BLAS: Design, implementation, and applications. *International Journal of High Performance Computing Applications (IJHPCA)*, 2011.

[36] A. Buluç, S. Williams, L. Oliker, and J. Demmel. Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication. In *Proc. IPDPS*, 2011.

[37] S. Carbon, A. Ireland, C. J. Mungall, S. Shu, B. Marshall, and S. Lewis. Amigo: online access to ontology and annotation data. *Bioinformatics*, 25(2):288–289, 2009.

[38] D. Chakrabarti, R. Kumar, and A. Tomkins. Evolutionary clustering. In *Proceedings of the ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, 2006.

[39] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A recursive model for graph mining. In *SIAM International Conference on Data Mining (SDM)*, 2004.

[40] 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*, APPROX '00, pages 84–95, 2000.

[41] M. S. Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the ACM International Symposium on Theory of Computing (STOC)*, 2002.

[42] J. Cheng, Y. Ke, S. Chu, and M. T. Ozsu. Efficient core decomposition in massive networks. In *IEEE International Conference on Data Engineering (ICDE)*, pages 51–62, 2011.

[43] Y. Chi, X. Song, D. Zhou, K. Hino, and B. L. Tseng. Evolutionary spectral clustering by incorporating temporal smoothness. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '07, pages 153–162, New York, NY, USA, 2007. ACM.

[44] N. Chiba and T. Nishizeki. Arboricity and subgraph listing algorithms. *SIAM J. Comput.*, 14:210–223, February 1985.

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

[46] J. Cohen. Graph twiddling in a MapReduce world. *Computing in Science & Engineering*, 11:29–41, 2009.

[47] M. Coscia, G. Rossetti, F. Giannotti, and D. Pedreschi. DEMON: A local-first discovery method for overlapping communities. In *Proceedings of the ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, 2012.

[48] Ö. Şimşek and A. G. Barto. Skill characterization based on betweenness. In *NIPS*, 2008.

[49] W. Cui, Y. Xiao, H. Wang, Y. Lu, and W. Wang. Online search of overlapping communities. In *SIGMOD*, 2013.

[50] L. Danon, A. Diaz-Guilera, and J. Duch. Comparing community structure identification. *Journal of Statistical Mechanics: Theory and Experiment*, 2005.

[51] DIMACS. 10th DIMACS implementation challenge. http://www.cc.gatech.edu/dimacs10.

[52] B. Doerr, M. Fouz, and T. Friedrich. Social networks spread rumors in sublogarithmic time. In *Proceedings of the 43rd annual ACM symposium on Theory of computing (STOC)*, pages 21–30, 2011.

[53] B. Doerr, M. Fouz, and T. Friedrich. Why rumors spread so quickly in social networks. *Communications of the ACM*, 55(6):70–75, June 2012.

[54] S. N. Dorogovtsev, A. V. Goltsev, and J. F. F. Mendes. k-core organization of complex networks. *Physical Review Letters*, 96, 2006.

[55] Y. Dourisboure, F. Geraci, and M. Pellegrini. Extraction and classification of dense communities in the web. In *World Wide Web Conference (WWW)*, pages 461–470, 2007.

[56] Y. Dourisboure, F. Geraci, and M. Pellegrini. Extraction and classification of dense communities in the web. In *Proc. of the 16th International Conference on World Wide Web*, WWW '07, pages 461–470, 2007.

[57] X. Du, R. Jin, L. Ding, V. E. Lee, and J. H. T. Jr. Migration motif: a spatial - temporal pattern mining approach for financial markets. In *Proc. of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '09, 2009.

[58] P. Erdős and A. Hajnal. On chromatic number of graphs and set-systems. *Acta Mathematica Hungarica*, 17:61–99, 1966.

[59] P. Erdös and A. Rényi. On the evolution of random graphs. In *Institute of Mathematics, Hungarian Academy of Sciences*, pages 17–61, 1960.

[60] U. Feige. Relations between average case complexity and approximation complexity. In *Proceedings of Symposium on Theory of Computing*, pages 534–543, 2002.

[61] D. R. Forsyth. *Group Dynamics*. Cengage Learning, 2010.

[62] S. Fortunato. Community detection in graphs. *Physics Reports*, 483(3-5):75–174, 2009.

[63] S. Fortunato. Community detection in graphs. *Physics Reports*, 486, 2010.

[64] E. Fratkin, B. T. Naughton, D. L. Brutlag, and S. Batzoglou. Motifcut: regulatory motifs finding with maximum density subgraphs. In *ISMB (Supplement of Bioinformatics)*, pages 156–157, 2006.

[65] L. Freeman. A set of measures of centrality based upon betweenness. *Sociometry*, 4, 1977.

[66] L. C. Freeman. Centered graphs and the structure of ego networks. *Mathematical Social Sciences*, 3(3), 1982.

[67] M. Gaertler. Dynamic analysis of the autonomous system graph. In *International Workshop on Inter-domain Performance and Simulation (IPS)*, pages 13–24, 2004.

[68] G. Gallo, M. D. Grigoriadis, and R. E. Tarjan. A fast parametric maximum flow algorithm and applications. *SIAM J. Comput.*, 18(1):30–55, Feb. 1989.

[69] R. Geisberger, P. Sanders, and D. Schultes. Better approximation of betweenness centrality. In *ALENEX*, 2008.

[70] C. Giatsidis, D. M. Thilikos, and M. Vazirgiannis. D-cores: Measuring collaboration of directed graphs based on degeneracy. In *IEEE International Conference on Data Mining (ICDM)*, pages 201–210, 2011.

[71] C. Giatsidis, D. M. Thilikos, and M. Vazirgiannis. Evaluating cooperation in communities with the k-core structure. In *International Conference on Advances in Social Network Analysis and Mining (ASONAM)*, pages 87–93, 2011.

[72] D. Gibson, R. Kumar, and A. Tomkins. Discovering large dense subgraphs in massive graphs. In *Proc. of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 721–732, 2005.

[73] A. Gionis, F. Junqueira, V. Leroy, M. Serafini, and I. Weber. Piggybacking on social networks. *Proc. VLDB Endow.*, 6(6):409–420, 2013.

[74] D. F. Gleich and C. Seshadhri. Vertex neighborhoods, low conductance cuts, and good seeds for local community methods. In *Proceedings of the ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, 2012.

[75] A. V. Goldberg. Finding a maximum density subgraph. Technical report, Berkeley, CA, USA, 1984.

[76] O. Green, R. McColl, and D. A. Bader. A fast algorithm for streaming betweenness centrality. In *Proceedings of ASE/IEEE SocialCom*, 2012.

[77] S. Gregory. Finding overlapping communities in networks by label propagation. *New Journal of Physics*, 12(10):103018, 2010.

[78] R. Gupta, T. Roughgarden, and C. Seshadhri. Decompositions of triangle-dense graphs. In *Innovations in Theoretical Computer Science (ITCS)*, pages 471–482, 2014.

[79] T. D. R. Hartley, Ü. V. Çatalyürek, A. Ruiz, F. Igual, R. Mayo, and M. Ujaldon. Biomedical image analysis on a cooperative cluster of GPUs and multicores. In *Proc. of the 22nd Annual International Conference on Supercomputing, ICS 2008*, pages 15–25, 2008.

[80] T. D. R. Hartley, A. R. Fasih, C. A. Berdanier, F. Özgüner, and Ü. V. Çatalyürek. Investigating the use of GPU-accelerated nodes for SAR image formation. In *Proc. of the IEEE International Conference on Cluster Computing, Workshop on Parallel Programming on Accelerator Clusters (PPAC)*, 2009.

[81] T. D. R. Hartley, E. Saule, and Ü. V. Çatalyürek. Improving performance of adaptive component-based dataflow middleware. *Parallel Computing*, 38(6-7):289–309, 2012.

[82] J. Healy, J. Janssen, E. Milios, and W. Aiello. Characterization of graphs using degree cores. In *Workshop on Algorithms and Models for the Web Graph (WAW)*, pages 137–148, 2006.

[83] K. Hildrum and P. Yu. Focused community discovery. In *Proceedings of the IEEE International Conference on Data Mining (ICDM)*, 2005.

[84] S. Hong, T. Oguntebi, and K. Olukotun. Efficient parallel graph exploration on multi-core CPU and GPU. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, 2011.

[85] J. Hopcroft and R. Tarjan. Algorithm 447: efficient algorithms for graph manipulation. *Communications of the ACM*, 16(6):372–378, June 1973.

[86] J. Håstad. Clique is hard to approximate within $n^{(1-\epsilon)}$. In *Acta Mathematica*, pages 627–636, 1996.

[87] H. Hu, X. Yan, Y. Huang, J. Han, and X. J. Zhou. Mining coherent dense subgraphs across massive biological networks for functional discovery. *Bioinformatics*, 21(1):213–221, Jan. 2005.

[88] L. Iasemidis, D.-S. Shiau, W. Chaovalitwongse, J. Sackellares, P. Pardalos, J. Principe, P. Carney, A. Prasad, B. Veeramani, and K. Tsakalis. Adaptive epileptic seizure prediction system. *Biomedical Engineering, IEEE Transactions on*, 50:616–627, 2003.

[89] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the ACM International Symposium on Theory of Computing (STOC)*, 1998.

[90] Y. Jia, V. Lu, J. Hoberock, M. Garland, and J. C. Hart. Edge vs. node parallelism for graph centrality metrics. In *GPU Computing Gems: Jade Edition*. 2011.

[91] R. Jin, Y. Xiang, N. Ruan, and D. Fuhry. 3-hop: a high-compression indexing scheme for reachability query. In *SIGMOD Conference*, pages 813–826, 2009.

[92] S. Jin, Z. Huang, Y. Chen, D. Chavarria-Miranda, J. Feo, and P. C. Wong. A novel application of parallel betweenness centrality to power grid contingency analysis. In *IPDPS*, 2010.

[93] S. Jin, Z. Huang, Y. Chen, D. G. Chavarría-Miranda, J. Feo, and P. C. Wong. A novel application of parallel betweenness centrality to power grid contingency analysis. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–7, 2010.

[94] G. J. Katz and J. T. Kider, Jr. All-pairs shortest-paths for large graphs on the GPU. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, GH '08, pages 47–55, Aire-la-Ville, Switzerland, Switzerland, 2008. Eurographics Association.

[95] S. Khot. Ruling out ptas for graph min-bisection, dense k-subgraph, and bipartite clique. *SIAM Journal on Computing*, 36(4):1025–1071, 2006.

[96] M.-S. Kim and J. Han. A particle-and-density based evolutionary clustering method for dynamic networks. *Proc. VLDB Endow.*, 2(1), Aug. 2009.

[97] S. Kintali. Betweenness centrality : Algorithms and lower bounds. *CoRR*, abs/0809.1906, 2008.

[98] G. Kortsarz and D. Peleg. Generating sparse 2-spanners. *Journal of Algorithms*, 17(2):222–236, 1994.

[99] D. Koschützki and F. Schreiber. Centrality analysis methods for biological networks and their application to gene regulatory networks. *Gene Regulation and Systems Biology*, 2, 2008.

[100] V. Krebs. Mapping networks of terrorist cells. *Connections*, 24, 2002.

[101] R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins. Trawling the web for emerging cyber-communities. In *Proc. of the Eighth International Conference on World Wide Web*, WWW '99, pages 1481–1493, 1999.

[102] A. Lancichinetti and S. Fortunato. Community detection algorithms: A comparative analysis. *Physical Review E*, 80, 2009.

[103] A. Lancichinetti, S. Fortunato, and J. Kertesz. Detecting the overlapping and hierarchical community structure in complex networks. *New Journal of Physics*, 2009.

[104] M.-J. Lee, J. Lee, J. Y. Park, R. H. Choi, and C.-W. Chung. QUBE: a Quick algorithm for Updating BEtweenness centrality. In *Proceedings of World Wide Web (WWW)*, 2012.

[105] V. E. Lee, N. Ruan, R. Jin, and C. C. Aggarwal. A survey of algorithms for dense subgraph discovery. In C. C. Aggarwal and H. Wang, editors, *Managing and Mining Graph Data*, volume 40. Springer, 2010.

[106] J. Leskovec, K. J. Lang, and M. Mahoney. Empirical comparison of algorithms for network community detection. In *Proceedings of the International World Wide Web Conference (WWW)*, 2010.

[107] R.-H. Li and J. X. Yu. Efficient core maintenance in large dynamic graphs. *CoRR*, abs/1207.4567, 2012.

[108] R. Lichtenwalter and N. V. Chawla. DisNet: A framework for distributed graph computation. In *ASONAM*, 2011.

[109] D. Lick and A. White. k-degenerate graphs. *Canadian Journal of Mathematics*, 22:1082–1096, 1970.

[110] Y.-R. Lin, Y. Chi, S. Zhu, H. Sundaram, and B. L. Tseng. Facetnet: A framework for analyzing communities and their evolutions in dynamic networks. In *Proceedings of the International World Wide Web Conference (WWW)*, 2008.

[111] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey. Efficient sparse matrix-vector multiplication on x86-based many-core processors. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, ICS '13, 2013.

[112] J.-K. Lou, S. d. Lin, K.-T. Chen, and C.-L. Lei. What can the temporal social behavior tell us? An estimation of vertex-betweenness using dynamic social information. In *ASONAM*, 2010.

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

[114] A. Lugowski, D. Alber, A. Buluç, J. Gilbert, S. Reinhardt, Y. Teng, and A. Waranis. A flexible open-source toolbox for scalable complex graph analysis. In *Proc. of SDM*, 2012.

[115] L. Luo, M. Wong, and W.-m. Hwu. An effective GPU implementation of breadth-first search. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 52–55, New York, NY, USA, 2010. ACM.

[116] K. Madduri, D. Ediger, K. Jiang, D. A. Bader, and D. G. Chavarria-Miranda. A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. In *IPDPS*, 2009.

[117] K. Matsumoto, N. Nakasato, and S. Sedukhin. Blocked all-pairs shortest paths algorithm for hybrid CPU-GPU system. In *High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on*, pages 145–152, 2011.

[118] D. Matula and L. Beck. Smallest-last ordering and clustering and graph coloring algorithms. *Journal of ACM*, 30(3):417–427, 1983.

[119] M. McGlohon, L. Akoğlu, and C. Faloutsos. *Statistical Properties of Social Networks*, chapter II in C. C. Aggarwal (ed.), Social Network Data Analytics. Springer, 2011.

[120] A. McLaughlin and D. A. Bader. Revisiting edge and node parallelism for dynamic GPU graph analytics. In *28th International Symposium on Parallel and Distributed Processing, Workshops and PhD Forum (IPDPSW), Workshop on Multithreaded Architectures and Applications (MTAAP)*, 2014.

[121] E. L. Merrer and G. Trédan. Centralities: Capturing the fuzzy notion of importance in social graphs. In *Proceedings of the Second ACM EuroSys Workshop on Social Network Systems (SNS)*, 2009.

[122] D. Merrill, M. Garland, and A. Grimshaw. Scalable GPU graph traversal. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 117–128, New York, NY, USA, 2012. ACM.

[123] S. M.Gabriella, A. Rossi, and C. Amici. Nf-kb and virus infection: who controls whom. *The EMBO Journal*, 11:2552–2560, 2003.

[124] D. Mizell and K. Maschhoff. Early experiences with large-scale Cray XMT systems. In *23rd International Symposium on Parallel and Distributed Processing, Workshops and PhD Forum (IPDPSW), Workshop on Multithreaded Architectures and Applications (MTAAP)*, pages 1–9, May 2009.

[125] R. Mokken. Cliques, clubs and clans. *Quality and Quantity*, 13(2):161–173, 1979.

[126] A. Muhammad. Graphs, simplicial complexes and beyond: Topological tools for multi-agent coordination. 2005.

[127] J. R. Munkres. *Elements of algebraic topology.* Addison-Wesley, 1984.

[128] A. A. Nanavati, G. Siva, G. Das, D. Chakraborty, K. Dasgupta, S. Mukherjea, and A. Joshi. On the structural properties of massive telecom call graphs: findings and implications. In *ACM International Conference on Information and Knowledge Management (CIKM)*, pages 435–444, 2006.

[129] A. A. Nanavati, G. Siva, G. Das, D. Chakraborty, K. Dasgupta, S. Mukherjea, and A. Joshi. On the structural properties of massive telecom call graphs: findings and implications. In *Proceedings of the ACM International Conference on Information and Knowledge Management (CIKM)*, 2006.

[130] M. E. J. Newman. Modularity and community structure in networks. *Proceedings of the National Academy of Sciences*, 103(23):8577–8582, 2006.

[131] T. Okuyama, F. Ino, and K. Hagihara. A task parallel algorithm for finding all-pairs shortest paths using the GPU. *Int. J. High Perform. Comput. Netw.*, 7(2):87–98, Apr. 2012.

[132] F. Ozgul, Z. Erdem, C. Bowerman, and C. Atzenbeck. Comparison of feature-based criminal network detection models with k-core and n-clique. In *International Conference on Advances in Social Network Analysis and Mining (ASONAM)*, pages 400–401, 2010.

[133] A. Padrol-Sureda, G. Perarnau-Llobet, J. Pfeifle, and V. Muntes-Mulero. Overlapping community search for social networks. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2010.

[134] G. Palla, I. Derenyi, I. Farkas, and T. Vicsek. Uncovering the overlapping community structure of complex networks in nature and society. *Nature*, 435:814–818, 2005.

[135] P. Pande and D. Bader. Computing betweenness centrality for small world networks on a GPU. Poster at 15th High Performance Embedded Computing Conference, Lexington, Massachusetts, 2011.

[136] M. C. Pham and R. Klamma. The structure of the computer science knowledge network. In *ASONAM*, 2010.

[137] R. Puzis, P. Zilberman, Y. Elovici, S. Dolev, and U. Brandes. Heuristics for speeding up betweenness centrality computation. In *SocialCom*, sep. 2012.

[138] U. N. Raghavan, R. Albert, and S. Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E*, 76, 2007.

[139] B. Rees and K. Gallagher. Overlapping community detection by collective friendship group inference. In *Proceedings of the International Conference on Advances in Social Network Analysis and Mining (ASONAM)*, 2010.

[140] R. A. Rossi, D. F. Gleich, A. H. Gebremedhin, and M. M. A. Patwary. A fast parallel maximum clique algorithm for large sparse graphs and temporal strong components. *CoRR*, abs/1302.6256, 2013.

[141] M. Rosvall and C. T. Bergstrom. Maps of random walks on complex networks reveal community structure. *Proceedings of the National Academy of Sciences*, 105(4):1118–1123, 2008.

[142] H. Saito, M. Toyoda, M. Kitsuregawa, and K. Aihara. A large-scale study of link spam detection by graph algorithms. In *International Workshop on Adversarial Information Retrieval on the Web (AIRWeb)*, pages 45–48, 2007.

[143] A. Sala, L. Cao, C. Wilson, R. Zablit, H. Zheng, and B. Y. Zhao. Measurement-calibrated graph models for social network experiments. In *WWW '10*, pages 861–870. ACM, 2010.

[144] R. Samudrala and J. Moult. A graph-theoretic algorithm for comparative modeling of protein structure. *Journal of Molecular Biology*, 279(1):287–302, 1998.

[145] A. E. Sarıyüce, B. Gedik, G. Jacques-Silva, K.-L. Wu, and Ü. V. Çatalyürek. Streaming algorithms for k-core decomposition. In *39th International Conference on Very Large Data Bases (VLDB)*, Aug 2013.

[146] A. E. Sarıyüce, B. Gedik, G. Jacques-Silva, K.-L. Wu, and Ü. V. Çatalyürek. Sonic: Streaming overlapping community detection. *Data Mining and Knowledge Discovery (DAMI) (under review)*, 2015.

[147] A. E. Sarıyüce, B. Gedik, G. Jacques-Silva, K.-L. Wu, and Ü. V. Çatalyürek. Streaming k-core decomposition: Algorithms and evaluation. *Very Large Data Bases Journal (VLDBJ) (under review)*, 2015.

[148] A. E. Sarıyüce, K. Kaya, E. Saule, and Ü. V. Çatalyürek. Betweenness centrality on gpus and heterogeneous architectures. In *Workshop on General Purpose Processing Using GPUs (GPGPU), in conjunction with ASPLOS*, Mar 2013.

[149] A. E. Sarıyüce, K. Kaya, E. Saule, and Ü. V. Çatalyürek. Incremental algorithms for closeness centrality. In *Proc of IEEE Int'l Conference on BigData*, Oct 2013.

[150] A. E. Sarıyüce, K. Kaya, E. Saule, and Ü. V. Çatalyürek. Graph manipulations for fast centrality computation. *Transactions on Knowledge Discovery from Data (TKDD) (under review)*, 2015.

[151] A. E. Sarıyüce, K. Kaya, E. Saule, and Ü. V. Çatalyürek. Incremental algorithms for network management and analysis based on closeness centrality. In *CoRR*, volume abs/1303.0422, 2013.

[152] A. E. Sarıyüce, E. Saule, K. Kaya, and Ü. V. Çatalyürek. Shattering and compressing networks for betweenness centrality. In *SIAM International Conference on Data Mining, SDM*, May 2013. An extended version is available as a Tech Rep on ¡a href="http://arxiv.org/abs/1209.6007"¿ArXiv¡/a¿.

[153] A. E. Sarıyüce, E. Saule, K. Kaya, and Ü. V. Çatalyürek. Streamer: a distributed framework for incremental closeness centrality computation. In *Proc of IEEE Cluster 2013*, Sep 2013.

[154] A. E. Sarıyüce, E. Saule, K. Kaya, and Ü. V. Çatalyürek. Hardware/software vectorization for closeness centrality on multi-/many-core architectures. In *28th International Symposium on Parallel and Distributed Processing, Workshops and PhD Forum (IPDPSW), Workshop on Multithreaded Architectures and Applications (MTAAP)*, May 2014.

[155] A. E. Sarıyüce, E. Saule, K. Kaya, and Ü. V. Çatalyürek. Incremental closeness centrality in distributed memory. *Parallel Computing*, 2015. to appear.

[156] A. E. Sarıyüce, E. Saule, K. Kaya, and Ü. V. Çatalyürek. Regularizing graph centrality computations. *Journal of Parallel and Distributed Computing*, 76(0):106 – 119, 2015. Special Issue on Architecture and Algorithms for Irregular Applications.

[157] A. E. Sarıyüce, C. Seshadhri, A. Pınar, and Ü. V. Çatalyürek. Finding the hierarchy of dense subgraphs using nucleus decompositions. In *24th International World Wide Web Conference (WWW)*, May 2015. Also available as a Tech Rep on ¡a href="http://arxiv.org/abs/1411.3312"¿ArXiv¡/a¿.

[158] E. Saule and Ü. V. Çatalyürek. An early evaluation of the scalability of graph algorithms on the Intel MIC architecture. In *26th International Symposium on Parallel and Distributed Processing, Workshops and PhD Forum (IPDPSW), Workshop on Multithreaded Architectures and Applications (MTAAP)*, May 2012.

[159] E. Saule, K. Kaya, and Ü. V. Çatalyürek. Performance evaluation of sparse matrix multiplication kernels on Intel Xeon Phi. In *Proc. of the 10th Int'l Conf. on Parallel Processing and Applied Mathematics (PPAM)*, Sep 2013.

[160] T. Schank and D. Wagner. Finding, counting and listing all triangles in large graphs, an experimental study. In *Experimental and Efficient Algorithms*, pages 606–609. Springer Berlin / Heidelberg, 2005.

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

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

[163] C. Seshadhri, A. Pinar, and T. G. Kolda. Triadic measures on graphs: The power of wedge sampling. *Statistical Analysis and Data Mining*, 7(4):294–307, 2014.

[164] Z. Shi and B. Zhang. Fast network centrality analysis using GPUs. *BMC Bioinformatics*, 12:149, 2011.

[165] J. Shun and G. E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, 2013.

[166] SNAP. Stanford network analysis package. http://snap.stanford.edu/snap.

[167] M. Sozio and A. Gionis. The community-search problem and how to plan a successful cocktail party. In *Proceedings of the ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, 2010.

[168] C. Stark, B.-J. Breitkreutz, T. Reguly, L. Boucher, A. Breitkreutz, and M. Tyers. Biogrid: a general repository for interaction datasets. *Nucleic Acids Research*, 34(Database-Issue):535–539, 2006.

[169] S. Suri and S. Vassilvitskii. Counting triangles and the curse of the last reducer. In *WWW'11*, pages 607–614, 2011.

[170] C. Tsourakakis, F. Bonchi, A. Gionis, F. Gullo, and M. Tsiarli. Denser than the densest subgraph: Extracting optimal quasi-cliques with quality guarantees. In *Proc. of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, 2013.

[171] C. E. Tsourakakis. A novel approach to finding near-cliques: The triangle-densest subgraph problem. *CoRR*, abs/1405.1477, 2014.

[172] D. Turaga, H. Andrade, B. Gedik, C. Venkatramani, O. Verscheure, J. D. Harris, J. Cox, W. Szewczyk, and P. Jones. Design principles for developing stream processing applications. *Software: Practice & Experience*, 40(12):1073–1104, 2010.

[173] Twitter. http://www.twitter.com/, retrieved March, 2012.

[174] A. Verma and S. Butenko. Network clustering via clique relaxations: A community based approach. *10th DIMACS Implementation Challenge*, 2011.

[175] R. Vuduc, J. Demmel, and K. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Proc. SciDAC 2005, J. of Physics: Conference Series*, 2005.

[176] F. Wang, T. Li, X. Wang, S. Zhu, and C. Ding. Community discovery using nonnegative matrix factorization. *Data Min. Knowl. Discov.*, 22(3), May 2011.

[177] J. Wang and J. Cheng. Truss decomposition in massive networks. *Proceedings of the VLDB Endowment*, 5(9):812–823, 2012.

[178] N. Wang, J. Zhang, K.-L. Tan, and A. K. H. Tung. On triangulation-based dense neighborhood graph discovery. *Proc. VLDB Endow.*, 4:58–68, 2010.

[179] S. Wasserman and K. Faust. *Social Network Analysis: Methods and Applications*. Cambridge University Press, 1994.

[180] D. Watts and S. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393:440–442, 1998.

[181] J. J. Whang, D. G. Gleich, and I. S. Dhillon. Overlapping community detection using seed set expansion. In *Proceedings of the ACM International Conference on Information and Knowledge Management (CIKM)*, 2013.

[182] S. Wuchty and E. Almaas. Peeling the yeast protein network. *PROTEOMICS*, 5(2):444–449, 2005.

[183] J. Xie, M. Chen, and B. K. Szymanski. LabelRankT: Incremental community detection in dynamic networks via label propagation. *CoRR*, abs/1305.2006, 2013.

[184] 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):43:1–43:35, Aug. 2013.

[185] B. Zhang and S. Horvath. A general framework for weighted gene co-expression network analysis. *Statistical Applications in Genetics and Molecular Biology*, 4(1):Article 17+, 2005.

[186] Y. Zhang and S. Parthasarathy. Extracting analyzing and visualizing triangle k-core motifs within networks. In *IEEE International Conference on Data Engineering (ICDE)*, pages 1049–1060, 2012.

[187] Y. Zhang and S. Parthasarathy. Extracting analyzing and visualizing triangle k-core motifs within networks. In *Proc. of the 2012 IEEE 28th International Conference on Data Engineering*, ICDE '12, pages 1049–1060, 2012.

[188] Y. Zhang and D.-Y. Yeung. Overlapping community detection via bounded nonnegative matrix tri-factorization. In *Proceedings of the ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, 2012.

[189] F. Zhao and A. K. H. Tung. Large scale cohesive subgraphs discovery for social network visual analysis. In *Proc. of the 39th international conference on Very Large Data Bases*, PVLDB'13, pages 85–96, 2013.